O'REILLY®

Java 9 口袋指南

Java Pocket Guide

[美] Robert Liguori Patricia Liguori 著
张卫滨 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书以通俗易懂的语言介绍了Java语言的语法、类型、并发编程等基础知识，同时还以样例的形式讲解了Lambda表达式、Java模块系统、JShell这些Java 8和Java 9新引入的特性。本书不仅能够满足初学者了解、掌握Java语言的需要，还能帮助资深的工程师快速熟悉和把握Java新技术的发展趋势。本书内容简洁、样例丰富，可以作为Java开发人员案头常备的参考书籍。

版权贸易合同登记号　图字：01-2017-8798

图书在版编目（CIP）数据

Java 9 口袋指南：第 4 版 /（美）罗伯特 · 利果里（Robert Liguori），（美）帕特丽夏 · 利果里（Patricia Liguori）著；张卫滨译 . —北京：电子工业出版社，2018.8

书名原文：Java Pocket Guide

ISBN 978-7-121-34602-6

Ⅰ . ① J… Ⅱ . ① 罗… ② 帕… ③ 张… Ⅲ . ① JAVA 语言－程序设计－指南

Ⅳ . ① TP312.8-62

中国版本图书馆 CIP 数据核字 (2018) 第 137790 号

责任编辑：张春雨

印　　刷：北京天宇星印刷厂

装　　订：北京天宇星印刷厂

出版发行：电子工业出版社

　　北京市海淀区万寿路173信箱　邮编：100036

开　　本：787×980　1/32　印张：8.125　字数：260千字

版　　次：2018年8月第1版

印　　次：2018年8月第1次印刷

定　　价：49.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

译者序

在程序员群体中，有两个未解之谜：那就是最好的编程语言和最好的编辑器到底是什么。关于语言的争论在程序员群体中是司空见惯的现象，至于哪个是世界上最好的编程语言也是见仁见智的，但不可否认的是 Java 在企业级和互联网开发中依然占据着重要的地位。除了“跨平台”这一特性早年带来的红利外，Java 能够 20 多年长盛不衰的原因在于它是一门不断演进和优化的语言。

尽管被其他语言的拥趸诟病演化缓慢，但 Java 确实在不断地革新，吸收和借鉴其他语言优秀的特征，比如 Lambda 表达式、函数式编程、泛型等。通过语言层面的改善，开发人员可以编写更加现代和更加简洁的代码，同时能够享受到软硬件架构体系演化所带来的收益。Java 另一个巨大的优势在于活跃的开源社区，像 Spring、Hibernate、Apache 等开源项目和组织，简化了大规模企业级 Java 应用的开发，抽离了技术底层的关注点，能够让我们专注于业务开发。因此，不管是 J2EE、SOA，还是近两年火热的微服务架构，都离不开 Java 语言的身影。

既然 Java 语言如此受欢迎，那么关于 Java 语言的技术图书用浩如烟海来形容就丝毫不过分了。在这方面既有专注于语言用法的经典图书，也有深入底层 JVM 原理的佳作，那么

这本《Java 9 口袋指南》的特殊之处在哪里呢？

这不是一本深入讲解 Java 语法细节的图书，也不是分析 JVM 实现原理的书，该书以 Java 的基本用法作为切入点，在介绍 Java 已有功能的基础上，重点讲解了 Java 8 和 Java 9 中的一些新特性，比如 Lambda 表达式、JShell、模块系统，能够让读者快速理解和掌握最新的技术。本书行文简洁，同时附带大量样例，能够让读者在示例中理解相关特性的原理和用法。

所谓“九层之台，起于累土”，我们只有掌握了 Java 的基本功能，夯实了知识基础，再去学习各种日新月异的开发框架，应对层出不穷的架构理念，才会得心应手。本书的英文版已经畅销多年，作者会根据 Java 语言的发展不断对内容进行更新和补充，希望中文版的发行能够帮助到更多的中国程序员朋友。

尽管在翻译的过程中，我力争达到准确和通畅，但限于水平和时间，肯定还有许多的不足或纰漏之处，热忱期待您提出意见，希望本书能够对您有用，您可以通过 levinzhang1981@126.com 联系到我，祝阅读愉快。

张卫滨

2018 年 5 月于大连

目录

前言 XIII

第 1 部分 语言

第 1 章 命名约定 3

缩略词 3
注解名 3
类名 4
常量名 4
枚举名称 4
泛型类型参数名 4
实例与静态变量名 5
接口名 5
方法名 5
包名 5
模块名 6
参数和本地变量名 6

第 2 章 词法元素 9

Unicode 与 ASCII 9
压缩字符串 11
注释 11
关键字 12
识别符 13
分隔符 14
操作符 15
字面量 16
转义序列 19
Unicode 货币符号 20

第 3 章 基本类型 23

原始类型 23
原始类型的字面量 24
浮点实体 26
原始类型的数值提升 28
包装类 29
自动装箱和拆箱 30

第 4 章 引用类型 33

引用类型与原始类型的对比 34
默认值 34
引用对象的转换 36
原始类型与引用类型的转换 37
传递引用类型到方法中 37
引用类型的对比 38
拷贝引用类型 40
引用类型的内存分配与垃圾回收 42

第 5 章 面向对象编程 43

类和对象 43
可变长度的参数列表 49
抽象类与抽象方法 50
静态数据成员、静态方法、静态常量以及静态初始化器 51
接口 52
枚举 53
注解类型 54
函数式接口 56

第 6 章 语句和代码块 57

表达式语句 57
空语句 58
代码块 58
条件语句 58
迭代语句 60
控制转移 62
synchronized 语句 63
断言语句 63
异常处理语句 64

第 7 章 异常处理 65

异常层级结构 65
检查型 / 非检查型异常和错误 66
常见的检查型 / 非检查型异常和错误 67
异常处理的关键字 69
异常处理的过程 74
定义自己的异常类 74
打印异常信息 75

第 8 章 Java 修饰符 77

访问修饰符 78
其他（非访问）修饰符 79
修饰符的编码 80

第 2 部分 平台

第 9 章 Java 平台，标准版 83

常用的 Java SE API 库 83

第 10 章 开发的基础工具 97

Java 运行时环境 97
Java 开发工具集 97
Java 程序结构 99
命令行工具 100
类路径 105

第 11 章 内存管理 107

垃圾收集器 107
内存管理工具 109
命令行参数 110
调整 Java 堆的大小 113
元空间 113
与 GC 进行交互 113

第 12 章 基本输入和输出 115

标准的流 in、out 和 err 115
标准输入和输出类的层级结构 116

文件读取和写入 117
Socket 读取和写入 118
序列化 120
压缩和解压文件 121

第 13 章 新 I/O API（NIO.2）..... 123

Path 接口 123
Files 类 124
其他特性 125

第 14 章 并发 127

创建线程 127
线程状态 128
线程优先级 129
常用方法 129
同步 130
并发工具集 132

第 15 章 Java 集合框架 135

Collection 接口 135
实现 136
集合框架方法 136
集合类的算法 137
算法的效率 138
Comparator 函数式接口 139
便利的工厂方法 142

第 16 章 泛型框架 143

泛型类与接口 143
具有泛型的构造器 144

替换原则 145
类型参数、通配符与边界 145
Get 和 Put 原则 146
泛型具体化 147
非泛型类型中的泛型方法 148

第 17 章 Java 脚本 API 149

脚本语言 149
脚本引擎实现 149
搭建脚本语言和引擎环境 151

第 18 章 日期和时间 API 155

与遗留系统的互操作 156
区域性日历 156
ISO 日历 156

第 19 章 Lambda 表达式 163

λE 基础 163
特定用途的函数式接口 166
通用的函数式接口 167
关于 λE 的资源 168

第 20 章 JShell：Java Shell 171

起步 171
片段 172
使用 JShell 173
JShell 的特性 180
JShell 命令小结 184

第 21 章 Java 模块系统.....187

Jigsaw 项目.....187
Java 模块.....188
编译模块.....190
模块化 JDK.....191
jdeps.....194
定义模块.....196
导出包.....196
声明依赖.....197
传递性依赖.....197
定义服务提供者.....198
jlink.....200

第 3 部分 附录

附录 A Fluent API.....203

附录 B 第三方工具.....205

附录 C UML 基础.....215

索引.....225

前言

《Java 9 口袋指南》一书的目标是成为读者的手边书，本书提供了 Java 语言和平台标准特性的快速指南。

《Java 9 口袋指南》提供了开发或调试 Java 程序所需的知识，包括了有用的编程样例、表格、图和列表。

本书中所介绍的 Java 是基于 Java SE 9 讲解的，包括了 80 个以上的 JDK 增强计划（JDK Enhancement Proposal，JEP）的子集。本书中介绍的 Java 包含了对语言的通用修改以及新的 Java Shell 和 Java 模块系统。本书将会取代之前的 *Java Pocket Guide*、*Java 7 Pocket Guide* 和 *Java 8 Pocket Guide*。

处于一致性和读者兴趣的考虑，第 4 版《Java 9 口袋指南》的主要代码对 Gliesians Web 应用（*http://gliesians.com/index-genealogy.faces*）的代码片段进行了更新。在编写本书的时候，Gliesians Web 应用的主要关注点在于提供免费的工具类，用于系谱学和小型无人机系统。

本书中还提供了准备 Oracle 认证程序员考试的资料。如果你考虑获取这样的 Java 证书，还可以购买 Edward Finegan 和 Robert Liguori 合著的 *OCA Java SE 8 Programmer I Study Guide* (*Exam 1Z0-808*)（McGraw-Hill Osborne Media, 2015）。

本书结构

本书分为三部分：第一部分详细介绍 Java 语言规范（JLS）和 JEP 所衍生的 Java 编程语言。第二部分介绍了 Java 平台组件和相关话题。第三部分是附录，介绍了相关的支撑技术。

本书中的约定

本书中使用了如下的排版约定：

斜体字（*Italic*）

表示新的术语、URL、Email 地址、文件名以及文件扩展名。

等宽字体（`Constant width`）

用于程序清单，以及在段落中对程序中元素的引用，如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

加粗的等宽字体（**`Constant width bold`**）

表示需要用户输入的命令或其他文本。

斜体的等宽字体（*`Constant width italic`*）

表示这些文本需要根据用户提供的值或上下文确定的值进行替换。

提示

这个元素代表提示、建议或一般说明。

警告

这个元素代表警告或提醒。

如何联系我们

如果您对本书有意见或问题，请联系出版社：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

我们提供了本书网页，上面列出了勘误表、示例和其他信息。请通过 *http://bit.ly/laravel-up-and-running* 访问该页。

要给出本书意见或者询问技术问题，请发送邮件到 *bookquestions@oreilly.com*。

更多有关书籍、课程、会议和新闻的信息，请见网站 *http://www.oreilly.com*。

在 Facebook 找到我们：*http://facebook.com/oreilly*
在 Twitter 上关注我们：*http://twitter.com/oreillymedia*
在 YouTube 上观看：*http://www.youtube.com/oreillymedia*

致谢

我要特别感谢 O'Reilly 的工作人员。感谢 Greg Grockenberger 和 Ryan Cuprak 的支持，他们分别撰写了 JShell 和 Java 模块化系统章节，还感谢 Ryan 担任了本书的技术审校。

我还要感谢最初参与 *Java Pocket Guide*、*Java 7 Pocket Guide* 和 *Java 8 Pocket Guide* 的人员。

特别感谢与本书的内容无关的一些人：Don Anderson、David Chong、Keith Cianfrani、Jay Clark、Steve Cullen、Ed DiCampli、Phil Greco、Scott Houck、Cliff Johnson、Juan Keller、Fran Kelly、Mike Krauss、Mike Lazlo、Phil Maloney、Lana Manovych、Matt Mariani、Chris Martino、Roe Morande、Sohrob Mottaghi、Brendan Nugent、Keith Smaniotto、Tom Tessitore、Lacey Thompson、Tyler Travis、Justin Trulear 和 Jack Wombough。

我最后还要感谢所有的家庭成员，感谢他们的陪伴。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/*

第 1 部分

语言

第 1 章

命名约定

命名约定能够让 Java 程序更具可读性。在编写程序时，使用含义明确无歧义的 Java 字符来组成各种名称是很重要的。如下的样例都来源于各种 Java 源码。

缩略词

如果在名称中使用缩略词，只有在适合使用大写字母的情况下，将缩略词的第一个字母改为大写的形式：

```
// 例如，DNA 表示为 Dna
public class GliesianDnaProvider {...}

// 例如，最近的共同祖先（Most Recent Common Ancestor，
MRCA）表示为 Mrca
public class MrcaCalculator {...}
```

注解名

Java SE API 已经为预定义的注解类型展现了多种声明注解名的方式，[形容词 | 动词] [名词]：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

类名

类名应该是名词，因为它们代表的是“事物”或“对象”。它们应该是大小写混合的（驼峰），每个单词的第一个字母应该大写，如下所示：

```
public class AirDensityCalculator {...}
```

常量名

常量名应该全是大写字母，多个单词之间通过下画线分割：

```
private static final double KELVIN = 273.16;
private static final double DRY_AIR_GAS_CONSTANT =
287.058;
private static final double HUMID_AIR_GAS_CONSTANT =
461.4964;
```

枚举名称

枚举名称应该遵循类命名的约定。枚举的对象集合（可选项）应该都是大写形式的：

```
public enum MeasurementSystem {
  METRIC, UNITED_STATES_CUSTOMARY, IMPERIAL
}

public enum Study {
  ALL, NON_ENDOGAMOUS, SEMI_ENDOGAMOUS, ENDOGAMOUS
}

public enum RelationshipMatchCategory {
  IMMEDIATE, CLOSE, DISTANT, SPECULATIVE
}
```

5 泛型类型参数名

泛型的类型参数名应该是一个大写字母。通常情况下，推荐使用 T 代指类型。

集合框架（Collections Framework）广泛使用了泛型。E 用来代指集合元素，S 用来代指服务加载器（service loader），K 和 V

分别用来代指 Map 中的键和值：

```
public interface Map <K,V> {
   V put(K key, V value);
}
```

实例与静态变量名

实例和静态变量名应该是名词，并且应该遵循与方法名相同的大小写约定：

```
private String prediction;
```

接口名

接口名应该是形容词。如果接口提供一定的功能，那么它应该以“able”或“ible”结尾；否则，它们应该是名词。接口名遵循与类名相同的大小写约定：

```
public interface Relatable {...}
public interface SystemPanel {...}
```

方法名

方法名应该包含一个动词，因为它们用于让某个对象采取一定的行为。它们应该是大小写混合的，开始是小写字母，随后的每个单词首字母大写。在方法名中可能会包含形容词或名词：

```
public void clear() {...} // 动词
public void toString() // 介词和名词
public double getDryAirDensity() {...} // 动词、形容词和名词
public double getHumidAirDensity() {...} // 动词、形容词和名词
```

6

包名

包名应该是唯一的并且要由小写字母组成，必要的时候可以使用下画线：

```
// Gliesian.com（公司），JAirDensity（软件）
package com.gliesian.jairdensity;

// Gliesian.com（公司），FOREX Calculator（软件），工具集
package com.gliesian.forex_calculator.utils;
```

公共可用的包应该按照该组织互联网域名的反向形式来命名，以顶级域名的单词作为开始（比如，com、net、org 或 edu），然后是组织的名字以及项目或分支的名字（内部包通常会按照项目来命名。）

以 java 或 javax 命名的包是受到严格限制的，只能用于为 Java 类库提供符合标准的实现。

模块名

模块名应该是反向的互联网域名，它和包名的规则是一样的：

```
module com.gliesian.utils {
}
```

参数和本地变量名

参数和本地变量名应该是描述性的小写单词、缩略语或缩写。如果需要多个单词，那么应该遵循方法名相同的约定：

```
public void printPredictions (ArrayList predictions) {
  int counter = 1;
  for (String prediction : predictions) {
    System.out.println("Predictions #" + counter++ + ":
" + prediction);
  }
}
```

临时变量应该由单个字母组成，比如整数所使用的 i、j、k、m 和 n，以及字符所使用的 c、d 和 e。临时变量和循环变量所对应的单字符常用名称如表 1-1 所示。

表1-1　临时变量和循环变量

单字符	类型
b	Byte
c	Character
d	Double
e	Exception
f	Float
i、j 或 k	Integer
l	Long
o	Object
s	String

第 2 章

词法元素

构成 Java 源码的单词或符号被称为*词法元素*或 *token*。Java 词法元素包括行终止符（line terminator）、空格、注释、关键字、识别符、分隔符、操作符和字面量。Java 程序语言中的单词和符号是由 Unicode 字符集组成的。

Unicode 与 ASCII

Unicode 是由 Unicode 协会（Unicode Consortium）标准组织维护的字符集，它的前 128 个字符与美国信息交换标准代码（American Standard Code for Information Interchange，ASCII）相同。Unicode 为每个字符提供了一个唯一的数字，能够跨所有的平台、程序和语言使用。Java SE 9 支持 Unicode 8.0.0。读者可以从在线手册（*http://www.unicode.org/versions/Unicode8.0.0/*）上了解 Unicode 标准的更多信息。Java SE 8 支持 Unicode 6.2.0。

提示

Java 注释、标识符以及字符串字面量并不局限于使用 ASCII 字符，而其他的 Java 输入元素都是由 ASCII 字符组成的。

关于特定 Java 平台所使用的 Unicode 字符集版本，在 Java API 的 Character 类中进行了文档化。用于脚本、符号和标点的 Unicode 字符编码表可以通过下面的网址查询：*http://unicode.org/charts/*。

可打印的 ASCII 字符

ASCII 预留代码 32（空格）以及代码 33~126（字母、数字、标点符号以及几个其他字符）作为可打印的字符。表 2-1 列出了这些代码数字值以及对应的 ASCII 字符。

表2-1　可打印的ASCII字符

```
32 SP 48 0 64 @ 80 P 96 ' 112 p
33 ! 49 1 65 A 81 Q 97 a 113 q
34 " 50 2 66 B 82 R 98 b 114 r
35 # 51 3 67 C 83 S 99 C 115 S
36 $ 52 4 68 D 84 T 100 d 116 t
37 % 53 5 69 E 85 U 101 e 117 u
38 & 54 6 70 F 86 V 102 f 118 v
39 ' 55 7 71 G 87 W 103 g 119 w
40 ( 56 8 72 H 88 X 104 h 120 x
41 ) 57 9 73 I 89 Y 105 i 121 y
42 * 58 : 74 J 90 Z 106 j 122 z
43 + 59 ; 75 K 91 [ 107 k 123 {
44 , 60 < 76 L 92 \ 108 l 124 |
11 45 - 61 = 77 M 93 ] 109 m 125 }
46 . 62 > 78 N 94 ^ 110 n 126 ~
47 / 63 ? 79 O 95 _ 111 o
```

不可打印的 ASCII 字符

ASCII 预留数字 0~31 和 127 作为控制字符。表 2-2 列出了这些代码的数字值以及对应的 ASCII 字符。

表2-2　不可打印的ASCII字符

00 NUL 07 BEL 14 SO 21 NAK 28 FS

01 SOH 08 BS 15 SI 22 SYN 29 GS

02 STX 09 HT 16 DLE 23 ETB 30 RS

03 ETX 10 NL 17 DC1 24 CAN 31 US

04 EOT 11 VT 18 DC2 25 EM 127 DEL

05 ENQ 12 NP 19 DC3 26 SUB

06 ACK 13 CR 20 DC4 27 ESC

> **提示**
>
> ASCII 10 是新行或换行，ASCII 13 是回车。

压缩字符串

压缩字符串特性是一种优化，它能够让字符串在内部表述时更节省空间。在 Java 9 中，该特性默认就是启用的。如果你主要使用 UTF-16 字符串，这项特性可以通过 `-XX:-CompactStrings` 标记来禁用。

注释

12

单行注释使用两个斜线开始，在遇到行终止字符时结束：

```
// 孩子的默认出生年份
private Integer childsBirthYear = 1950;
```

多行注释以斜线和星号开始，以星号和斜线作为结束。在它们

之间每行使用单个星号作为格式约定，这是通常的用法，但并不是必需的：

```
/*
* 2001 年，美国妇女生孩子的平均年龄是 24.9 岁，所以我们使用 25
* 作为默认值。
*/
private Integer mothersAgeGivingBirth = 25;
```

Javadoc 注释会由 Javadoc 工具进行处理，生成 HTML 格式的 API 文档。Javadoc 必须要以斜线和两个星号作为开始，以斜线和一个星号作为结束（Oracle 的文档页面，*http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.htm]* 提供了 Javadoc 工具的更多信息）：

```
/**
 * 父母年龄的预测器 *
 * @author Robert J. Liguori
 * @author Gliesian, LLC.
 * @version 0.1.1 09-02-16
 * @since 0.1.0 09-01-16
 */
public class GenitorBirthdatePredictorBean {...}
```

在 Java 中，注释不能嵌套：

```
/* This is /* not permissible */ in Java */
```

13 关键字

表 2-3 中包含了 Java 9 的关键字。其中的 `const` 和 `goto` 关键字是预留的，在 Java 语言中并没有使用它们。

提示

在 Java 程序中，Java 关键字不能用作标识符。

表2-3 Java关键字

abstract	enum	module	synchronized
assert	exports	native	this
boolean	extends	new	throw
break	final	package	throws
byte	finally	private	to
case	float	protected	transient
catch	for	provides	try
char	goto	public	uses
class	if	requires	void
const	implements	return	volatile
continue	import	short	while
default	instanceof	static	with
do	int	strictfp	_
double	interface	super	
else	long	switch	

提示

有时候，true、false和null字面量被误认为是关键字。它们并不是关键字，而是预留的字面量。

14

识别符

Java 字面量是程序员给类、方法、变量等的名称。

识别符不能与任意的关键字、boolean 或 null 字面量具有相同的 Unicode 字符序列。

Java 识别符是由 Java 字母（Java letter）组成的，而 Java 字母指的是 Character.isJavaIdentifierStart(int) 字符能够返回 `true` 的字符。Java 字母由 ASCII 字符集组成，但是限于美元符号（$）、大小写的字母以及下画线（_）。注意，在 Java 9 中，（_）是一个关键字，不能单独作为标识符。

在第一个字符之后，允许使用数字作为标识符：

```
// 合法识别符的样例
class GedcomBean {
  private File uploadedFile;  // 大写和小写字母
  private File _file; // 以下画线开头
  private File $file; // 以 $ 开头
  private File file1; // 非数字作为开头
}
```

可以参阅第 1 章了解命名指南。

分隔符

有一些 ASCII 字符会将程序分割为多个部分，它们被用作分隔符。()、{ }、[ ] 和 < > 都是成对使用的：

```
() { } [ ] < > :: : ; , . ->
```

表 2-4 列出了这些括号的名称，可以使用这些名称代指不同类
15 型的括号分隔符。在名称列表中的第一个通常也会用在 Java 语言规范之中。

表2-4　Java的括号分隔符

括号	名称	用途
()	小括号或圆括号，英文可以对应称为 Parentheses、curved brackets、oval brackets 和 brackets	在算数表达式中调整优先级，类型转换及封装方法的参数

续表

括号	名称	用途
{}	大括号或花括号，英文可以对应称为 Braces、curly brackets、fancy brackets、squiggly brackets 和 squirrelly brackets	封装代码块并用于实现数组功能
[]	方括号或中括号，英文可以对应为 Box brackets、closed brackets 或 square brackets	实现数组功能和数组的初始化
<>	尖括号或菱形括号，英文可以对应为 Angle brackets、diamond brackets 和 chevrons	封装泛型

书名符号，又称尖括号引用（angle quotes），即 << >>，用于 UML 中的原型（stereotype）。

操作符

操作符会针对一个、两个或三个操作数执行一定的操作并返回结果。在 Java 中操作符类型包括赋值、计算、比较、位操作、递增 / 递减以及类 / 对象。表 2-5 按照优先顺序列出了 Java 的操作符（最优先的放到了表格的顶部），同时给出了操作符的概要描述和结合性（associativity，从左到右或从右到左）。

表2-5　Java的操作符

16

优先级	操作符	描述	结合性
1	++、--	后递增、后递减	R → L
2	++、--	前递增、前递减	R → L
	+、-	一元（unary）加、一元减	R → L
	~	按位取反（Bitwise complement）	R → L
	!	布尔非	R → L

续表

优先级	操作符	描述	结合性
3	new	创建对象	R → L
	(type)	类型转换	R → L
4	*、/、%	乘、除、求余	L → R
5	+、-	加、减	L → R
	+	字符串连接	L → R
6	<<、>>、>>>	左移、右移、无符号右移	L → R
7	<、<=、>、>=	小于、小于等于、大于、大于等于	L → R
	instanceof	类型对比	L → R
8	==、!=	值相等或不相等	L → R
	==、!=	引用相等或不相等	L → R
9	&	布尔与	L → R
	&	位与	L → R
10	^	布尔异或（XOR）	L → R
	^	位异或（XOR）	L → R
11	|	布尔或	L → R
	|	位或	L → R
12	&&	逻辑与（又称，条件性与）	L → R
13	||	逻辑或（又称，条件性或）	L → R
14	?:	条件性三元运算符	L → R
15	=、+=、-=、	赋值操作符	R → L
	*=、/=、%=、		
	&=、		
	^=、|=、<<=、		
	>> =、>>>=		

字面量

字面量是值的源码表述。从 Java SE 开始，在数字字面量中允许使用下画线以增加代码的可读性。下画线只能放在数字之间，

在运行时会被忽略。

关于原始类型字面量的更多信息，请参见第 3 章“原始类型的字面量”章节。

布尔字面量

布尔字面量要使用 true 和 false 来表达：

```
boolean isFullRelation = true;
boolean isHalfRelation = Boolean.valueOf(false); //
unboxed
boolean isEndogamyPresent = false;
```

字符字面量

字符字面量要么是一个字符，要么是一个转义序列（escape sequence），它要包含在单引号中，其中不允许是行终止符：

```
char charValue1 = 'a' ;
// 撇号
Character charValue2 = Character.valueOf( '\'' );
```

整型字面量

18

整数类型（byte、short、int 和 long）可以采用十进制、十六进制、八进制和二进制来表示。在默认情况下，整数字面量是 int 类型的：

```
int intValue1 = 34567, intValue2 = 1_000_000;
```

十进制整数可以包含任意的 ASCII 数字，即 0 到 9，它代表的是正数：

```
Integer integerValue1 = Integer.valueOf(100);
```

为数字添加一个一元否定操作符作为前缀能够形成一个负十进制数字：

```
public static final int INT_VALUE = -200;
```

十六进制字面量以 0x 或 0X 开始，随后是 ASCII 数字 0 到 9 以及 a 到 f（或者 A 到 F）的字母。在处理十六进制字面量时，

Java 并不区分大小写。

十六进制数字可以表示正整数、负整数和零：

```
int intValue3 = 0X64; // 十六机制的数字 100
```

八进制数字以零作为开始，随后紧接着一个或多个从 0 到 7 的 ASCII 数字：

```
int intValue4 = 0144; // 八进制的数字 100
```

二进制字面量使用 0b 或 0B 作为前缀，随后紧接着的是 0 和 1：

```
char msgValue1 = 0b01001111; // O
char msgValue2 = 0B01001011; // K
char msgValue3 = 0B0010_0001; // !
```

要将一个整数定义为 long 类型，在后面添加 ASCII 字符 L 或 l（优先推荐使用前者，它更易读）：

```
long longValue = 100L;
```

19

浮点字面量

一个合法的浮点字面量需要整数部分和 / 或小数部分、小数点以及类型后缀。由 e 或 E 表示的指数是可选的。在使用指数形式或带有类型后缀时，小数部分和小数点并不是必需的。

浮点字面量（double）是一个八字节的双精度浮点，float 是四个字节。double 类型的后缀是 d 或 D，float 类型的后缀是 f 或 F：

```
[whole-number].[fractional_part][e|E exp][f|F|d|D]

float floatValue1 = 9.15f, floatValue2 = 1_168f;
Float floatValue3 = new Float(20F);
double doubleValue1 = 3.12;
Double doubleValue2 = Double.valueOf(1e058);
float expValue1 = 10.0e2f, expValue2=10.0E3f;
```

字符串字面量

字符串字面量包含零个或多个字符，其中也可以包含转义序列，

它们会放到双引号中。字符串字面量不能包含 Unicode \u000a 或 \u000d 作为行终止符，请使用 \r 或 \n 替代。字符串常量是不可变的：

```
String stringValue1 = new String( “Valid literal.” );
String stringValue2 = “Valid.\nOn new line.” ;
String stringValue3 = “Joins str” + “ings” ;
String stringValue4 = “\” Escape Sequences\” \r” ;
```

类 String 会有一个关联的字符串池。初始时，这个池是空的。字符串字面量和以字符串为值的常量表达式都会插入到这个池中，并且只会放入一次。

如下样例阐述了字面量是如何添加到池中以及如何使用这个池的：

```
// 添加 “thisString” 到池中
String stringValue5 = “thisString” ;
// 使用池中的 “thisString” 字符串
String stringValue6 = “thisString” ;
```


通过调用字符串的 intern() 方法，字符串可以添加到池中（如果池中还不存在）。intern() 方法会返回一个字符串，它可能是对添加到池中的新字符串的引用，也可能是对已有字符串的引用：

```
String stringValue7 = new String( “thatString” );
String stringValue8 = stringValue7.intern();
```

Null 字面量

Null 字面量是 null 类型，能够用于引用类型。它无法用到原始类型上：

```
String n = null;
```

转义序列

表 2-6 列出了 Java 中的转义序列。

表2-6　转义序列的字符和字符串字面量

名称	序列	十进制	Unicode
退格	\b	8	\u0008
水平制表键	\t	9	\u0009
换行	\n	10	\u000A
换页	\f	12	\u000C
回车	\r	13	\u000D
双引号	\”	34	\u0022
单引号	\’	39	\u0027

不同平台会有不同的行终止符（参见表 2-7）。`println()` 方法会包含换行，这比直接硬编码使用 `\n` 和 `\r` 会更好一些。

21 表2-7　不同平台的新行变种

操作系统	新行
兼容 POSIX 的操作系统（如 Solaris、Linux）和 macOS	LF (\n)
Microsoft Windows	CR+LF (\r\n)
macOS 9 版本以下	CR (\r)

Unicode 货币符号

Unicode 货币符号的范围是 \u20A0 到 \u20CF（8352–8399），参见表 2-8 的示例。

表2-8　货币符号

名称	符号	十进制	Unicode
法郎符号	₣	8355	\u20A3
里拉符号	₤	8356	\u20A4
米尔符号	₥	8357	\u20A5
卢比符号	Rs	8360	\u20A8
盾符号	₫	8363	\u20AB
欧元符号	€	8364	\u20AC
德拉克马符号	₯	8367	\u20AF
德国便士符号	₰	8368	\u20B0

还有一些符号不在预设的范围之内，参见表 2-9。

表2-9　范围外的货币符号

22

名称	符号	十进制	Unicode
美元符号	$	36	\u0024
美分符号	¢	162	\u00A2
英镑符号	£	163	\u00A3
货币符号	¤	164	\u00A4
日元符号	¥	165	\u00A5
（带钩的拉丁文小写字母 F）Latin small f with hook	ƒ	402	\u0192
孟加拉卢比马克	৲	2546	\u09F2
孟加拉卢比符号	৳	2547	\u09F3
古吉拉特语卢比符号	૱	2801	\u0AF1
泰米尔卢比符号	௹	3065	\u0BF9
泰国泰铢符号	฿	3647	\u0E3F
手写体大写 M	ℳ	8499	\u2133
中日韩统一表意文字（CJK Unified Ideograph）1	元	20803	\u5143
中日韩统一表意文字 2	円	20870	\u5186
中日韩统一表意文字 3	圆	22278	\u5706
中日韩统一表意文字 4	圓	22291	\u5713

第 3 章

基本类型

基本类型包括 Java 的原始类型及其对应的封装类 / 引用类型。借助自动装箱（autoboxing）和拆箱（unboxing）功能，Java 提供了原始类型和引用类型之间的自动转换。在适当的情况下，会使用数字提升（numeric promotion）。

原始类型

在 Java 中，有 8 种原始类型，每种都是保留的关键字。这些原始类型描述了具有对应格式和尺寸的单个值（参见表 3-1）。不管底层的硬件精度如何（比如 32 位或 64 位），原始类型都会具有固定的精度。

表3-1　原始类型

类型	描述	存储	范围
boolean	true 或 false	1 比特	不适用
char	Unicode 字符	2 字节	\u0000 到 \uFFFF
byte	整型	1 字节	–128 到 127
short	整型	2 字节	–32768 到 32767
int	整型	4 字节	–2147483648 到 2147483647

24

续表

类型	描述	存储	范围
long	整型	8 字节	–263 到 263 –1
float	浮点	4 字节	1.4e–45 到 3.4e+38
double	浮点	8 字节	5e–324 到 1.8e+308

提示

原始类型 byte、short、int、long、float 和 double 是有符号的，char 类型是没有符号的。

原始类型的字面量

除了 boolean 以外，其他的原始类型都可以接受字符、十进制、十六进制、八进制、Unicode 文字格式以及字符转义序列。在合适的地方，字面值会自动转型或转换。需要记住的是，在截断（truncation）的过程会有比特的丢失。下面展现了一系列原始类型赋值的样例。

boolean 类型合法的字面量只有 true 和 false：

```
boolean isEndogamous = true;
```

char 类型代表了一个 Unicode 字符。超过两个字节的 char 类型的字面量值需要显式地进行转型。

```
// 'atDNA'
  char[] cArray = {
  '\'' , // '
  '\u0061', // a
  't' , // t
  0x0044, // D
  0116, // N
  (char) (65 + 131072) , // A
  0b00100111}; // '
```

25 byte 原始类型以 4 个字节的有符号整数作为合法的字面量。如果没有进行明确的转型，整数会隐式地转型为一个字节：

```
final byte CHROMOSOME_PAIRS = 12;
final byte CHROMOSOME_TOTAL = (byte) 48;
```

short 原始类型以 4 个字节的有符号整数作为合法的字面量。如果没有进行明确的转型，整数会隐式地转型为两个字节：

```
short firstCousins = 6;
short secondCousins = (short) 18;
```

int 原始类型以 4 个字节的有符号整数作为合法的字面量。当使用 char、byte 和 short 原始类型作为字面量的时候，它们会自动转型为 4 个字节的整数。浮点型和 long 类型的字面量必须要进行显式的转型：

```
int thirdCousins = 104;
int forthCousins = (int) 648.0D;
int fifthCousins = (short) 3_888;
```

long 原始类型以 8 个字节的有符号整数作为合法的字面量，这是通过 L 或 l 后缀来指定的。如果没有后缀或者使用了转型，那么值将会从 4 个字节转换为 8 个字节：

```
long sixthCousins = 23_000;
long seventhCousins = (long) 138_000;
long eighthCousins = 828_000l;
long ninthCousins = 4_968_000L;
```

float 原始类型以 4 个字节的有符号浮点数作为合法的字面量，这是通过 F 或 f 后缀或显式的类型转换来指定的。尽管对于整型字面量来说，明确的类型转换并不是必需的，但是当值的范围超过 223 的时候，int 值并不一定总是适合转换为 float：

```
float totalSharedCentimorgansX = 0;
float totalSharedCentimorgansAutosomal = (float) 285.5;  <26>
float largestSharedCentimorgansX = 0.0f;
float largestSharedCentimorgansAutosomal = 71F;
```

double 原始类型以 8 个字节的有符号浮点数作为合法的字面量，这是通过 D 或 d 后缀或无后缀的显式类型转换来指定的。如果字面量为整数，它会进行隐式的类型转换：

```
double centimorgansSharedFloor = 0;
```

```
double centimorgansSharedCeiling = 6766.20;
double centimorgansShared = (double) 888;
double centimorgansUnShared = 5878.0d;
double centimorgansPercentShared = 13.12D;
```

关于字面量的更多细节，请参考第 2 章。

浮点实体

正负浮点无穷大、负零以及非数字（not a number，NaN）都是为了满足 IEEE 754-1985 标准定义的特殊实体（参见表 3-2）。

如果某个操作所创建的浮点数的值太大，无法用传统的方式来表示，那么就会返回 Infinity、–Infinity 或 –0.0 实体。

表3-2　浮点实体

实体	描述	样例
`Infinity`	代表了正无穷的概念	`1.0 / 0.0`、`1e300 / 1e-300`、`Math.abs (-1.0 / 0.0)`
`-Infinity`	代表了负无穷的概念	`-1.0 / 0.0`、`1.0 / (-0.0)`、`1e300/-1e- 300`
`-0.0`	代表了一个接近零的负数	`-1.0 / (1.0 / 0.0)`、`-1e-300 / 1e300`
`NaN`	代表了未定义的结果	`0.0 / 0.0`、`1e300 * Float.NaN`、`Math.sqrt (-9.0)`

27 正无穷、负无穷和 NaN 实体可以以 `double` 和 `float` 常量的形式来获取：

```
Double.POSITIVE_INFINITY; // Infinity
Float.POSITIVE_INFINITY;  // Infinity
Double.NEGATIVE_INFINITY; // -Infinity
Float.NEGATIVE_INFINITY;  // -Infinity
Double.NaN; // Not-a-Number
Float.NaN;  // Not-a-Number
```

`Double` 和 `Float` 封装类都有方法判断一个数字是有限的数、无穷大还是 NaN：

```
Double.isFinite(Double.POSITIVE_INFINITY); // false
Double.isFinite(Double.NEGATIVE_INFINITY); // false
Double.isFinite(Double.NaN); // false
Double.isFinite(1); // true
// true
Double.isInfinite(Double.POSITIVE_INFINITY);
// true
Double.isInfinite(Double.NEGATIVE_INFINITY);
Double.isInfinite(Double.NaN); // false
Double.isInfinite(1); // false
Double.isNaN(Double.NaN); // true
Double.isNaN(1); // false
```

特殊实体相关的操作

表 3-3 展示了特殊实体操作的结果，其中 Double.POSITIVE_INFINITY 简写为 INF，Double.NEGATIVE_INFINITY 简写为 –INF，Double.NaN 简写为 NaN。

例如，表中第 4 列的表头（–0.0）以及第 12 行的条目（* NAN）所形成的结果是 NaN，可以写成如下的形式：

```
// 将会打印“NaN”
System.out.print((-0.0) * Double.NaN);
```

表3-3　特殊实体相关的操作

28

	INF	(-INF)	(-0.0)
* INF	Infinity	-Infinity	NaN
+ INF	Infinity	NaN	Infinity
– INF	NaN	-Infinity	-Infinity
/ INF	NaN	NaN	–0.0
* 0.0	NaN	NaN	–0.0
+ 0.0	Infinity	–Infinity	0.0
+ 0.5	Infinity	–Infinity	0.5
* 0.5	Infinity	–Infinity	–0.0
+ (–0.5)	Infinity	–Infinity	–0.5
* (–0.5)	–Infinity	–Infinity	0.0
+ NAN	NAN	NaN	NaN
* NAN	NAN	NaN	NaN

提示

对 NaN 执行的任意操作形成的结果都是 NaN，不存在 -NaN 的概念。

原始类型的数值提升

数值提升（numeric promotion）包含了在一定条件下对操作数进行数字运算的规则。数值提升规则包括一元和二元的规则。

29

一元数值提升

当一个原始的数字类型是表 3-4 中的表达式的一部分时，就会应用如下的提升规则：

- 如果操作数是 `byte`、`short` 或 `char` 类型，那么类型将会提升为 `int`。
- 否则，操作数的类型保持不变。

表3-4　一元数值提升规则的表达式

表达式
一元加操作符的操作数
一元减操作符（-）的操作数
按位取反操作符（~）的操作数
所有的移动操作符（>>、>>> 或 <<）
数组访问表达式中的索引表达式
数组创建表达式中的维度表达式

二元数值提升

当两个不同数字类型的原始值进行表 3-5 所列的运算时，其中一个类型将会按照如下的二进制提升规则进行数值提升：

- 如果其中一个类型为 `double`，非 `double` 类型的原始类型会转换为 `double` 类型；

- 如果其中一个类型为 float,非 float 类型的原始类型会转换为 float 类型。
- 如果其中有一类型为 long，非 long 类型的原始类型会转换为 long 类型；
- 否则，两个操作数都会被转换为 int 类型。

表3-5　二元提升规则的操作符

30

操作符	描述
+ 和 –	加法操作符
*、/ 和 %	乘法操作符
<、<=、> 和 >=	对比操作符
== 和 !=	相等操作符
&、^ 和 |	位操作符
? :	条件操作符（参见下一节）

条件操作符的特例

- 如果一个操作数为 byte 类型，另一个操作数为 short 类型，那么条件表达式将会是 short 类型的：

```
short = true ? byte : short
```

- 如果一个操作数 R 为 byte、short 或 char 类型，另一个操作数为 int 类型，并且它的类型在 R 的范围之内，那么条件表达式是 R 类型的：

```
short = (true ? short : 1967)
```

- 否则，将会应用二元数值提升规则，条件表达式的类型将会是第二个和第三个操作数提升之后的类型。

包装类

每个原始类型都有一个对应的包装类 / 引用类型，它们位于 java.lang 包中。每个包装类都有各种方法，包括返回类型值的方法，如表 3-6 所示。其中整型和浮点型的包装类可以返回多个原始类型的值。

31 表3-6 包装类

原始类型	引用类型	获取原始类型值的方法
boolean	Boolean	booleanValue()
char	Character	charValue()
byte	Byte	byteValue()、shortValue()、intValue()、longValue()、floatValue()、doubleValue()
short	Short	byteValue()、shortValue()、intValue()、longValue()、floatValue()、doubleValue()
int	Integer	byteValue()、shortValue()、intValue()、longValue()、floatValue()、doubleValue()
long	Long	byteValue()、shortValue()、intValue()、longValue()、floatValue()、doubleValue()
float	Float	byteValue()、shortValue()、intValue()、longValue()、floatValue()、doubleValue()
double	Double	byteValue()、shortValue()、intValue()、longValue()、floatValue()、doubleValue()

自动装箱和拆箱

自动装箱和拆箱一般用于原始类型的集合。自动装箱涉及动态分配内存以及为每个原始类型的值创建对象。注意，它的损耗通常有可能超过所需操作的执行时间。拆箱涉及为每个对象创建一个原始类型的值。

计算密集型的任务在使用原始类型时（比如在容器中通过原始
32 类型进行遍历），应该使用原始类型的数组而不是包装对象的集合。

自动装箱

自动装箱指的是自动将原始类型转换为对应的包装类。在本例中，每个物种的二倍体染色体数目（如 60、46 和 38）自动转换为对应的包装类，这是因为集合存储的是引用，而不是原始值：

```
// 创建权重组的 HashMap
HashMap<String, Integer> diploidChromosomeNumberMap
    = new HashMap<String, Integer> ();
diploidChromosomeNumberMap.put( “Canis latrans” , 78);
diploidChromosomeNumberMap.put( “Bison bison” , 60);
diploidChromosomeNumberMap.put( “Homo sapiens” , 46);
diploidChromosomeNumberMap.put( “Sus scrofa” , 38);
diploidChromosomeNumberMap.put( “Myrmecia pilosula” ,
2);
```

如下的样例展现了一个可以接受但是并不推荐的自动装箱用法：

```
// 设置常染色体的数量（atDNA）
Integer atDnaChromosomeSet = 22; // 不恰当的用法
```

因为没有必要进行强制自动装箱，上面的语句应该写成如下形式：

```
Integer atDnaChromosomeSet = Integer.valueOf(22);
```

拆箱

拆箱指的是自动将包装类转换为对应的原始类型。在本例中，会从 HashMap 中返回一个引用类型，它会自动拆箱，所以能够适用于原始类型：

```
// 设置智人的 DCN，执行自动拆箱 int homoSapienDcn =
diploidChromosomeNumberMap.get( “Homo sapiens” );

System.out.println(homoSapienDcn);
$ 46
```

33

如下的样例展现了可以接受但是并不推荐的自动拆箱用法：

```
// 设置染色体的总数
Integer atDnaChromosomeSet = 22;
int multiplier = 2;
int xChromosomes = 2; // 1 or 2
int yChromosome = 0;  // 0 or 1
```

```
// 混合 int 和 Integer，不推荐
int dcn = xChromosomes + yChromosome
  + (multiplier * atDnaChromosomeSet);
```

更好的做法是使用 intValue() 方法来撰写表达式，如下所示：

```
int dcn = xChromosomes + yChromosome
  + (multiplier * atDnaChromosomeSet.intValue());
```

第 4 章

引用类型

引用类型持有对象的引用，并且提供了一种方式来访问位于内存某处的这些对象。内存分配与程序员无关。所有的引用类型都是 java.lang.Object 类型的子类。

表 4-1 列出了 Java 的 5 种引用类型

表4-1　引用类型

引用类型	简述
注解（Annotation）	提供了一种方式将元数据（关于数据的数据）与编程元素关联起来
数组（Array）	提供了大小固定的数据结构，该数据结构会存放类型相同的数据元素
类（Class）	设计用来提供继承、多态以及封装功能。通常用于为现实世界中的某些东西建模，包含了一组值和一组方法，其中值会持有数据，而方法用来操作这些数据
枚举（Enumeration）	对一组对象的引用，这组对象代表了相关的可选值
接口（Interface）	提供公开 API，它需要被 Java 类来“实现（implement）”

36 引用类型与原始类型的对比

在 Java 中有两种类型：引用类型和原始类型。表 4-2 展示了它们之间的一些核心差异。参见第 3 章了解更多的细节。

表4-2 引用类型与原始类型的对比

引用类型	原始类型
引用类型的数量不受限制，因为它们是由用户定义的	包括 `boolean` 和数字类型：`char`、`byte`、`short`、`int`、`long`、`float` 和 `double`
内存分配会存储数据的引用	内存分配会存储原始类型持有的实际数据
当一个引用类型赋值给另外一个引用类型时，它们都会指向同一个对象	当一个原始类型的值赋值给另外一个相同类型的变量时，会生成值的一个副本
当对象传递给一个方法时，被调用的方法可以修改传递过来的对象内容，而不是修改对象的地址	当一个原始值传递给一个方法时，传递的只是原始值的副本。被调用的方法不能访问最初的原始值，因此无法对其进行修改。被调用的方法可以修改得到的副本的值

默认值

在 Java 中，默认值指的是当没有明确设置初始化值的时候，赋值给实例变量的值。

实例与本地变量对象

实例变量（也就是声明在类级别的变量）有个默认值为 null。null 不引用任何内容。

本地变量（也就是在方法内声明的变量）没有默认值，甚至连 null 值都没有。要始终记得初始化本地变量，因为它们没有设置默认值。检查未初始化本地变量对象是否为某个值（包括是

否为 null 值）将会导致编译错误。

尽管值为 null 的对象引用不关联堆中的任意对象，但是将对象 <37>
设置为 null 之后，就不会出现编译期和运行时的错误：

```
LocalDate birthdate = null;
// 这将会编译通过
if (birthdate == null) {
  System.out.println(birthdate);
}
$ null
```

如果引用变量为 null，调用它的方法或者使用点号操作符将会产生 java.lang.NullPointerException：

```
final int MAX_LENGTH = 20;
String partyTheme = null;
/*
 * 将会抛出 java.lang.NullPointerException，因为
partyTheme 为 null
*/
if (partyTheme.length() > MAX_LENGTH) {}
```

数组

不管是声明为实例变量还是本地变量，数组始终都会给定一个默认值。已声明但没有初始化的数组会给定一个默认值 null。

在下面的代码中，gameList1 数组进行了初始化，但是数组中的每个值尚未初始化，这意味着对象引用会是 null。对象需要添加到数组之中：

```
/*
 * 声明为 gameList1 和 gameList2 的数组已被初始化，默认为 null
  */
Game[] gameList1;
Game gameList2[];

/*                                                          <38>
 * 如下的数组已被初始化，但是其中的对象引用依然为 null，因为数
组不包含对象
 */
   gameList1 = new Game[10];

// 添加 Game 到列表中，所以它具有一个对象
```

```
gameList1[0] = new Game();
```

在 Java 中，多维数组实际上是数组的数组。它们可以使用 new 操作符初始化或者通过将值放到大括号中来初始化。多维数组在形式上可以是一致的，也可以是不一致的：

```
// 匿名数组
int twoDimensionalArray[][] = new int[6][6];
twoDimensionalArray[0][0] = 100;
int threeDimensionalArray[][][] = new int[2][2][2];
threeDimensionalArray[0][0][0] = 200;
int varDimensionArray[][] = {{0,0},{1,1,1},
{2,2,2,2}};
varDimensionArray[0][0] = 300;
```

匿名数组允许我们在代码中的任意位置创建数组值：

```
// 使用匿名数组的例子
int[] luckyNumbers = new int[] {7, 13, 21};
int totalWinnings = sum(new int[] {3000, 4500,
5000});
```

引用对象的转换

一个对象可以转换为其超类类型（扩大转换）或者其子类类型（收缩转换）。

编译器会在编译时检查转换，Java 虚拟机（Java Virtual Machine，JVM）也会在运行时进行转换的检查。

扩大转换

39

- 扩大转换（widening）指的是隐式地将子类转换为父类（超类）。
- 扩大转换不会抛出运行时异常。
- 没有必要进行显式的类型转换：

```
String s = new String();
Object o = s; // 扩大转换
```

收缩转换

- 收缩转换（narrowing）指的是将一个更通用的类型转换为更

具体的类型。

- 收缩是将超类转换为子类。
- 需要显式地进行类型转换。要将某个对象转换为另一个对象时，在被转换的对象前面以圆括号的形式给出想要转换成的目标对象类型。
- 不合法的收缩转换会导致 ClassCastException。
- 收缩转换可能会导致丢失数据 / 精确度。

对象不能转换为毫不相关的类型——也就是无法转换成不是其子类或超类的类型。如果这样做，将会在编译期产生不能转换的类型（inconvertible types）错误。如下就是一个类型无法转换的编译期错误样例：

```
Object o = new Object();
String s = (Integer) o;  // 编译错误
```

40

原始类型与引用类型的转换

原始类型与引用类型之间相互的自动类型转换被称为自动装箱和拆箱。关于这方面的更多信息，请参见第 3 章。

传递引用类型到方法中

当一个对象以变量的形式传递到一个方法中的时候：

- 传递的是引用变量的副本，而不是实际的对象。
- 调用方法和被调方法具有相同的引用副本。
- 调用方法能够看到被调方法对对象所做的任何变更。将对象的副本传递给被调方法能够避免对原始对象的变更。
- 被调方法无法改变对象的地址，但是它可以改变对象的内容。

如下的样例阐述了如何传递引用类型和原始类型到被调方法中，以及被调方法改变这些类型后所带来的影响：

```
void roomSetup() {
  // 引用传递
  Table table = new Table();
```

```
    table.setLength(72);
    // 长度将会发生变化
    modTableLength(table);

    // 原始类型传递
    // 椅子的数量不会变化
    int chairs = 8;
    modChairCount(chairs);
  }

41 void modTableLength(Table t) {
    t.setLength(36);
  }

  void modChairCount(int i) {
    i = 10;
  }
```

引用类型的对比

在 Java 中，引用类型是可以对比的。相等操作符和 `equals` 方法可以用来协助进行对比。

使用相等操作符

相等操作符 != 和 == 能够用来对比两个对象的内存位置。如果要对比的对象的内存地址相同，那么对象就认为是相同的。这些相等操作符不会用于比较两个对象的内容。

在下面的样例中，guest1 和 guest2 具有相同的内存地址，所以将会输出“They are equal”语句：

```
String guest1 = new String( “name” );
String guest2 = guest1;
if (guest1 == guest2)
  System.out.println( “They are equal” );
```

在下面的样例中，内存地址并不相同，所以将会输出“They are not equal”：

```
String guest1 = new String( “name” );
String guest2 = new String( “name” );
if (guest1 != guest2)
  System.out.println( “They are not equal” );
```

使用 equals() 方法

42

要比较两个类对象的内容，可以使用或覆盖 Object 类所定义的 equals() 方法。当覆盖 equals() 方法时，hashCode() 方法也需要被覆盖。这样做是为了兼容基于 hash 的集合，比如 HashMap() 和 HashSet()。

> **提示**
>
> 在默认情况下，equals() 只会使用 == 操作符进行对比。这个方法要想真正有用的话，需要覆盖它。

例如，如果你想比较同一个类的两个实例中的值，那么就需要使用程序员所定义的 equals() 方法。

比较字符串

在 Java 中，有两种方式来比较字符串是否相等，但是每种方法对于“相等”的定义是不同的：

- equals() 方法会逐字符比较两个字符串，以此来决定它们是否相等。这并不是 Object 类默认的 equals() 方法实现，它是 String 类覆盖后的实现。
- == 操作符检查两个对象引用是否引用了对象的同一个实例。

下面的程序阐述了使用 equals() 方法和 == 操作符是如何评估字符串的（关于字符串如何评估的更多信息，请参见第 2 章“字符串字面量”一节）：

```
class MyComparisons {

  // 将字符串添加到池中                                    43
  String first = “chairs”;
  // 使用池中的字符串
  String second = “chairs”;
  // 创建新的字符串
  String third = new String ( “chairs”);
```

```
 void myMethod() {

  /*
   * 可能与常规的想法相反，这会判断为 true，读者可以尝试一下!
   */
  if (first == second) {
    System.out.println( “first == second” );
  }

  // 判断为 true
  if (first.equals(second)) {
    System.out.println( “first equals second” );
  }
  // 判断为 false
  if (first == third) {
    System.out.println( “first == third” );
  }
  // 判断为 true
  if (first.equals(third)) {
    System.out.println( “first equals third” );
  }
 } // End myMethod()
} //end class
```

提示

StringBuffer 和 StringBuilder 类的对象是可变的，String 类的对象是不可变的。

44 比较枚举

enum 值可以使用 == 操作符或 equals() 方法进行对比，因为它们返回的值是相同的。在对比枚举类型时，== 操作符会用得更加频繁。

拷贝引用类型

当拷贝引用类型时，有两种方式，第一种会生成对象引用的一个副本，第二种会创建一个新对象，生成实际对象的副本。在 Java 中，后者被称为克隆（clone）。

将引用拷贝到一个对象上

当拷贝对象的引用时，所形成的结果是一个对象有了两个引用。在下面的样例中，closingSong 被赋值成了 lastSong 所指向的对象引用。任何对 lastSong 所做的变更都会反映到 closingSong 上，反之亦然：

```
Song lastSong = new Song();
Song closingSong = lastSong;
```

克隆对象

克隆会形成对象的另外一个副本，而不仅仅是对象引用的副本。类默认是不能克隆的。需要注意，克隆通常会非常复杂，所以应该考虑使用一个副本构造器作为替代方案，原因如下：

- 类要想支持克隆，就要实现 Cloneable 接口；
- 受保护的 clone() 方法允许对象克隆自身；
- 一个对象如果想要克隆非自身的其他对象，被克隆的对象需要覆盖 clone() 方法，并且要对外公开；
- 在进行克隆的时候，必须要进行类型转换，因为 clone() 返回的是 Object 类型；45
- 克隆可能会抛出 CloneNotSupportedException。

浅克隆和深克隆

在 Java 中，存在浅克隆和深克隆两种克隆类型。

在浅克隆中，被克隆对象的原始值和引用会生成副本。这些引用所指向的对象不会生成副本。

在下面的样例中，leadingSong 会被赋予 length 的值，因为它是原始类型。leadingSong 会被赋予对 title、artist 和 year 的引用，因为它们是引用类型：

```
Class Song {
  String title;
  Artist artist;
  float length;
```

```
  Year year;
  void setData() {...}
}
Song firstSong = new Song();
try  {
  // 通过克隆进行实际的拷贝
  Song leadingSong = (Song)firstSong.clone();
} catch (CloneNotSupportedException cnse) {
  cnse.printStackTrace();
} // end
```

在深克隆中，被克隆对象会生成其每个字段的副本，这个过程中会递归它所引用的所有其他对象。深克隆方法必须由程序员来定义，因为 Java API 中并没有提供。深克隆的替代方案是序列化和副本构造器（通常情况下，更加推荐使用副本构造器，而不是序列化）。

46 引用类型的内存分配与垃圾回收

当创建新的对象时，会分配内存。当一个对象没有引用的时候，对象所使用的内存可以在垃圾收集过程中回收。关于这个话题的更多信息，请参见第 11 章。

第 5 章

面向对象编程 47

在 Java 中面向对象编程（*object-oriented programming*，OOP）的基本元素是类、对象和接口。

类和对象

类所定义的实体通常代表了现实世界中的事物。它们由一组值和一组方法构成，其中值会持有数据，而方法则会操作这些数据。

类可以从其他的类继承数据成员和方法。一个类只能直接继承另一个类，也就是其超类（superclass）。一个类只能有一个直接超类，这称为继承（inheritance）。

类的实例被称为对象，对象会分配内存。一个类可以有多个实例。

当实现类的时候，类的内部细节应该是私有的（private），只能通过公开的接口来访问。这称为封装（encapsulation）。JavaBean 约定会使用访问器和赋值（accessor 和 mutator）方法（如 getFirstName() 和 setFirstName(“Leonardina”)）来直接访问类的私有成员，从而确保其他的类不会意外修改私有成员。返回不可变值（如字符串、原始类型值以及精心设计的不可变对象）是另外一种防止数据成员被其他对象修改的方式。 48

类语法

类由类签名、可选的构造器、数据成员和方法构成：

```
[javaModifiers] class className
[extends someSuperClass]
[implements someInterfaces separated by commas] {
  // 数据成员
  // 构造器
  // 方法
}
```

实例化类（创建对象）

对象是类的实例。初始化之后，对象就有自己的数据成员和方法：

```
// 示例类的定义
public class Candidate {...}
class Stats extends ToolSet {...}


public class Report extends ToolSet
  implements Runnable {...}
```

通过使用 new 关键字，类 Candidate 会创建（初始化）多个对象：

```
Candidate candidate1 = new Candidate();
Candidate candidate2 = new Candidate();
```

数据成员和方法

49 数据成员，也被称为*字段*，会持有类的数据。非静态的数据成员也被称为*实例变量*：

```
[javaModifier] type dataMemberName
```

方法会操作类数据：

```
[javaModifiers] type methodName (parameterList)
[throws listOfExceptionsSeparatedByCommas] {
// 方法体
}
```

如下是样例类 Candidate 及其数据成员和方法：

```
public class Candidate {
```

```
  // 数据成员或字段
  private String firstName;
  private String lastName;
  private String party;
  // 方法
  public void setParty (String p) {party = p;}
  public String getLastName() {return lastName;}
} // End class Candidate
```

访问对象中的数据成员和方法

点操作符（.）用来访问对象中的数据成员和方法。在对象内部，访问数据成员和方法时，不需要使用点操作符：

```
candidate1.setParty( "Whig" );
String name = getFirstName() + getLastName();
```

重载

方法（包括构造器）都能进行重载。重载指的是两个或更多的方法具有相同的名称，但是签名不同（参数和返回值）。需要注意，重载方法必须具有不同的参数，返回类型可能不相同，但是仅仅返回类型不相同并不是重载。重载方法的访问修饰符可以不相同：

```
public class VotingMachine {
  ...
  public void startUp() {...}
  private void startUp(int delay) {...}
}
```

50

当方法重载时，允许每个签名有抛出不同的检查型异常：

```
private String startUp(District d) throws IOException
{...}
```

覆盖

子类可以覆盖它所继承的方法。在实现覆盖的时候，方法要具备和超类相同的签名（名称和参数），但是它会包含不同的实现细节。

Display 类中的 startUp() 方法，在 TouchScreenDisplay 类中

进行了覆盖：

```
public class Display {
  void startUp(){
    System.out.println( “Using base display.” );
  }
}
public class TouchScreenDisplay extends Display {
    void startUp() {
    System.out.println( “Using new display.” );
  }
}
```

方法覆盖包含如下的规则：

- 非 final、private 或 static 的方法才能进行覆盖；
- protected 方法可以覆盖没有访问修饰符的方法；
- 覆盖方法的访问修饰符（如 package、public、private、protected）不能比原始方法更严格；
- 覆盖方法不能抛出任何新的检查型异常。

51

构造器

构造器在对象创建的时候会被调用，它用于初始化新创建对象的数据。构造器是可选的，它与类具有完全一致的名称，在方法体中，它不能（像方法那样）具有 return 语句。

类可以有多个构造器。当新对象创建时，与签名相匹配的构造器会被调用：

```
public class Candidate {
  ...
  Candidate(int id) {
    this.identification = id;
  }
  Candidate(int id, int age) {
    this.identification = id;
    this.age = age;
  }
}
// 创建一个新的 Candidate 并调用其构造器
Candidate candidate = new Candidate(id);
```

如果没有明确声明构造器，类都会有一个隐式的无参构造器。

需要注意的是，如果我们添加了有参数的构造器，那么无参构造器就不存在了，除非我们手动添加进来。

超类和子类

在 Java 中，类（称为*子类*）可以直接继承另一个类（称为*超类*）。Java 关键字 `extends` 表明类会从另外一个类继承数据成员和方法。子类不能直接访问超类的 `private` 成员，但是能够访问超类的 `public` 和 `protected` 成员。子类还能访问超类包共享的成员（*包私有*或 `protected`）。如之前所述，`accessor` 和 `mutator` 方法提供了间接访问类的 `private` 成员的机制，其中也包括超类的成员：

52

```
public class Machine {
  boolean state;
  void setState(boolean s) {state = s;}
  boolean getState() {return state;}
}
public class VotingMachine extends Machine {
  ...
}
```

关键字 `super` 能够用来访问超类中那些被子类覆盖掉的方法：

```
public class PrivacyWall {
  public void printSpecs() {
    System.out.println( “Printed PrivacyWall Specs” );
  }
}
public class Curtain extends PrivacyWall {
  public void printSpecs() {
    super.printSpecs();
    System.out.println( “Printed Curtain Specs” );
  }
  public static void main(String[] args) {
    Curtain curtain = new Curtain();
    curtain.printSpecs();
  }
}
$ Printed PrivacyWall Specs
$ Printed Curtain Specs
```

关键字 `super` 的另外一种常见用法就是调用超类的构造器并传递参数。需要注意的是，这个对 `super` 的调用必须是构造器中

的第一条语句：

```
53    public PrivacyWall(int length, int width) {
        this.length = length;
        this.width = width;
        this.area = length * width;
      }

      public class Curtain extends PrivacyWall {
        // 设置默认的长度和宽度
        public Curtain()  {super(15, 25);}
      }
```

如果没有显式调用超类的构造器，那么它会自动调用超类的无参构造器。

this 关键字

this 关键字的 3 个常见用法是引用当前对象、在某个构造器中引用同一个类中的另外一个构造器以及将当前对象的引用传递给另外的对象。

将参数变量赋值给当前对象的实例变量：

```
public class Curtain extends PrivacyWall {
  String color;
  public void setColor(String color) {
    this.color = color;
  }
}
```

在某个构造器中调用同一个类中的另外一个构造器：

```
public class Curtain extends PrivacyWall {
  public Curtain(int length, int width) {}
  public Curtain() {this(10, 9);}
}
```

54 将当前对象的应用传递给另外一个对象：

```
// 打印 Curtain 类的内容
System.out.println(this);

public class Builder {
  public void setWallType(Curtain c) {...}
}
```

可变长度的参数列表

方法可以具有可变长度的参数列表，它们被称为 vararg 参数，这种方法在调用时，最后一个参数（只能是最后一个）可以重复出现零次或多次。vararg 参数可以是原始类型或对象。方法在声明 vararg 时，要在方法的参数列表中使用省略符号（...）。vararg 参数的语法如下所示：

type... objectOrPrimitiveName

如下是 vararg 方法签名的一个样例：

```
public setDisplayButtons(int row,
 String... names) {...}
```

Java 编译器会将 vararg 方法修改为常规的方法，上面的样例在编译期会修改为：

```
public setDisplayButtons(int row,
 String [] names) {...}
```

vararg 方法允许将 vararg 参数作为唯一的参数：

```
// 零行或多行
public void setDisplayButtons (String... names)
{...}
```

vararg 方法与常规方法的调用是一样的，只不过它能接受可变数量的参数，允许在最后一个参数上重复设置：

```
setDisplayButtons( “Jim” );
setDisplayButtons( “John” , “Mary” , “Pete” );
setDisplayButtons( “Sue” , “Doug” , “Terry” , “John” );
```

55

printf 方法通常用来格式化可变集合的输出，因为 printf 是一个 vararg 方法。在 Java API 中，它的签名如下所示：

```
public PrintStream printf(String format,
  Object... args)
```

printf 方法在调用时会接受一个格式化字符串以及一个可变的对象集合：

```
System.out.printf( “Hello voter %s%n
  This is machine %d%n” , “Sally” , 1);
```

至于如何格式化传入到 printf 方法中的字符串的详细信息，可以参见 java.util.Formatter 类。

增强的 for 循环（for each）通常会用来迭代可变参数：

```
printRows() {
  for (String name: names)
    System.out.println(name);
}
```

抽象类与抽象方法

抽象类和抽象方法要使用 abstract 关键字来声明。

抽象类

抽象类一般用来作为基类并且不能实例化。它可以包含抽象和非抽象方法，抽象类可以作为抽象类或非抽象类的子类。如果子类不是抽象类，那么它所继承（扩展）的所有非抽象方法必须要进行定义：

```
public abstract class Alarm {
  public void reset() {...}
  public abstract void renderAlarm();
}
```

56

抽象方法

抽象方法只包含方法声明，它必须由某个继承它的非静态类定义：

```
public class DisplayAlarm extends Alarm {
  public void renderAlarm() {
    System.out.println( “Active alarm.” );
  }
}
```

静态数据成员、静态方法、静态常量以及静态初始化器

静态数据成员、方法、常量以及初始化器（initializer）是与类共生的，而不是与类的实例共存。静态数据成员、方法和常量可以通过定义它们的类来访问，也可以在其他类中通过点操作符来访问。

静态数据成员

静态数据成员具有和静态方法相同的特性，存储在内存中一个单一位置中。

它们的适用场景就是类的所有实例只能有该数据成员的一个副本（比如计数器）：

```
// 声明静态的数据成员
public class Voter  {
  static int voterCount = 0;
  public Voter() { voterCount++;}
  public static int getVoterCount() {
    return voterCount;
  }
}
...
int numVoters = Voter.voterCount;
```

57

静态方法

静态方法在声明的时候具有关键字 `static`：

```
// 声明静态方法
class Analyzer {
  public static int getVotesByAge() {...}
}
// 使用静态方法
Analyzer.getVotesByAge();
```

静态常量

静态常量就是声明为静态的常量。它们具有关键字 `static` 和

final，程序不能对它们进行修改：

```
// 声明静态常量
static final int AGE_LIMIT = 18;
// 使用静态常量
if (age == AGE_LIMIT)
  newVoter = "yes";
```

静态初始化器

静态初始化器包含一段以 static 关键字为前缀的代码。一个类可以具有任意数量的静态初始化代码块，类能够保证它们会按照出现的顺序执行。静态初始化器只会在类初始化之前执行一遍。代码块是在 JVM 类加载器加载类的时候执行的，也就是对代码初始引用的时候：

```
public class Election {
  private static int numberOfCandidates;
  // 静态初始化器
  static {
    numberOfCandidates = getNumberOfCandidates();
  }
}
```

58

接口

接口提供了一组声明为 public 的方法，这些方法不包含方法体。类实现某个接口的时候，它要么为接口中定义的所有方法提供具体实现，要么将类声明为抽象类。

接口使用关键字 interface 来声明，后面跟着接口的名称和一组方法声明。

接口名称通常是形容词，并且以“able”或“ible”结尾，因为接口提供了某种能力：

```
interface Reportable  {
  void genReport(String repType);
  void printReport(String repType);
}
```

类实现接口时必须要指明，这需要在其类签名中使用关键字

implements：

```
class VotingMachine implements Reportable {
  public void genReport (String repType) {
    Report report = new Report(repType);
  }
  public void printReport(String repType) {
    System.out.println(repType);
  }
}
```

在 Java 8 中，我们可以在接口中提供方法的实现。Java 9 则引入了私有接口方法。

提示

类可以实现多个接口，接口也可以扩展多个其他的接口。

枚举

59

按照最简单的形式，枚举就是一组对象，代表了可选值的集合：

```
enum DisplayButton {ROUND, SQUARE}
DisplayButton round = DisplayButton.ROUND;
```

如果从最简单的形式再深入了解一下，枚举是 enum 类型的类，并且它是单例的。枚举类可以有方法、构造器和数据成员：

```
enum DisplayButton {
   // 按照英寸所定义的尺寸
   ROUND (.50f),
   SQUARE (.40f);
   private final float size;
   DisplayButton(float size) {this.size = size;}
   private float size()  { return size; }
}
```

values() 方法能够返回一个有序的对象数组，其中包含了枚举所定义的对象：

```
for (DisplayButton b : DisplayButton.values())
```

```
System.out.println( “Button: “ + b.size());
```

注解类型

注解提供了一种将元数据（关于数据的数据）与程序元素在编译期和运行期关联起来的方式。包、类、方法、字段、参数、变量以及构造器都可以添加注解。

内置的注解

Java 注解提供了获取类元数据的方式。Java 有多个内置的注解类型，如表 5-1 所示。这些注解类型包含在 java.lang 包中。

60 内置的注解必须直接放到被标注的元素前面，它们不会抛出异常。注解能够返回原始类型、枚举、`String` 类、`Class` 类、注解以及（对应类型的）数组。

表5-1　内置注解

注解类型	描述
`@Override`	表明该方法要覆盖超类中的方法
`@Deprecated`	表明正在使用或覆盖废弃的 `API`。`Java 9` 添加了 `forRemoval` 和 `since` 方法
`@FunctionalInterface`	定义有且仅有一个抽象方法
`@SafeVarargs`	编码人员的断言，表明被注解的方法或构造器体中不会对 `varargs` 参数执行不安全的操作
`@SuppressWarnings`	用来有选择性地抑制警告信息

如下阐述了注解使用的样例：

```
@Deprecated(forRemoval=true)
  public void method () {
    ;
  }

@Override
  public String toString() {
```

```
    return super.toString() + " more";
  }
```

因为 @Override 是一个标记（marker）注解，所以如果无法找到要被覆盖的方法，那么将会返回编译警告。

开发人员定义的注解

开发人员可以使用 3 种注解类型定义自己的注解。标记（marker）注解没有参数，单值（single value）注解有一个参数，多值（multivalue）注解有多个参数。

61

注解的定义由 @ 符号、interface 关键字以及注解名字组成。

注解允许重复出现。

元注解 Retention 表明注解能够被 VM 所获取，所以它能够在运行时读取到。Retention 位于 java.lang.annotation 包中：

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Feedback {} //标记
public @interface Feedback {
  String reportName();
} //单值
public @interface Feedback {
  String reportName();
  String comment() default "None";
} //多值
```

我们可以将用户定义的注解直接放到被标注的条目上：

```
@Feedback(reportName="Report 1")
public void myMethod() {...}
```

程序可以针对某个方法调用 getAnnotation() 方法判断注解是否存在并获取注解的值：

```
Feedback fb =
  myMethod.getAnnotation(Feedback.class);
```

类型注解规范（Type Annotations Specification，也被称为 JSR 308）允许将注解放到数组和泛型参数上。注解还可以用于超类、所实现的接口、类型转换、instanceof 检查、异常规范、通

配符、方法引用以及构造器引用上。参见 Cay S. Horstmann 所编写 *Java SE 8 for the Really Impatient* 以了解注解用于这些场景的详细信息。

62 函数式接口

函数式接口，也被称为*单抽象方法*（single abstract method，SAM）接口，是仅仅定义一个抽象方法的接口。注解 @FunctionalInterface 可以放到接口的前面，用来表明它是函数式接口。接口可以有任意数量的默认方法：

```
@FunctionalInterface
  public interface InterfaceName {

  // 只允许有一个抽象方法
  public void doAbstractTask();

  // 允许有多个默认方法
  default public void performTask1(){
    System.out.println( “Msg from task 1.” );
  }
  default public void performTask2(){
    System.out.println( “Msg from task 2.” );
  }
}
```

函数式接口的实例可以通过 lambda 表达式、方法引用或构造器引用来创建。

第 6 章

语句和代码块 63

语句是单条命令，当它被 Java 解释器执行时，会完成某种活动。

```
GigSim simulator = new GigSim( “Let’s play guitar!” );
```

Java 语句包含如下类型：表达式、空语句、代码块、条件、迭代、控制转移、异常处理、变量（variable）、标记（labeled）、断言和同步。

在语句中使用的保留字包括 if、else、switch、case、while、do、for、break、continue、return、synchronized、throw、try、catch、finally 和 assert。

表达式语句

表达式语句是修改程序状态的语句。Java 表达式以分号作为结尾。表达式语句包括赋值、前后递增、前后递减、对象创建以及方法调用。如下是表达式语句的样例：

```
isWithinOperatingHours = true;
++fret; patron++; --glassOfWater; pick--;
Guitarist guitarist = new Guitarist();
guitarist.placeCapo(guitar, capo, fret);
```

64

空语句

空语句不会提供任何的额外功能，它可以写为一个分号（;）或空的代码块 {}。

代码块

一组语句被称为*块*或*代码块*。代码块会封装到花括号中，声明在代码块中的变量和类分别称为*局部变量*和*局部类*。局部变量和局部类的作用域（scope）就是声明它们的代码块。

在代码块中，语句按照它们的编写顺序或流程控制的顺序解释执行。如下是一个代码块的样例：

```
static {
  GigSimProperties.setFirstFestivalActive(true);
  System.out.println( “First festival has begun” );
  gigsimLogger.info( “Simulator started 1st
festival” );
}
```

条件语句

`if`、`if else`以及`if else if`是决策判断流控制语句。它们用于有条件地执行语句，这些语句表达式必须是`Boolean`或`boolean`类型，`Boolean`会借助拆箱机制，自动将`Boolean`转换为`boolean`。

if 语句

`if`语句有一个表达式、一个语句或一个语句块所组成，如果表达式的执行结果为 true，语句或语句块就会执行：

65

```
Guitar guitar = new Guitar();
guitar.addProblemItem( “Whammy bar” );
if (guitar.isBroken()) {
  Luthier luthier = new Luthier();
  luthier.repairGuitar(guitar);
}
```

if else 语句

当 else 与 if 结合使用时，如果表达式的执行结果为 true，第一个语句块的代码会执行，否则将会执行 else 中的代码块：

```
CoffeeShop coffeeshop = new CoffeeShop();
if (coffeeshop.getPatronCount() > 5) {
  System.out.println( “Play the event.” );
} else {
  System.out.println( “Go home without pay.” );
}
```

if else if 语句

当我们需要从多个代码块之间做出选择时，通常会用到 if else if 语句。当断言不匹配所有的代码块时，将会执行最终 else 中的代码块：

```
ArrayList<Song> playList = new ArrayList<>();
Song song1 = new Song( “Mister Sandman” );
Song song2 = new Song( “Amazing Grace” );
playList.add(song1);
playList.add(song2);
...
int numOfSongs = playList.size();
if (numOfSongs <= 24) {
  System.out.println( “Do not book” );
} else if ((numOfSongs > 24) & (numOfSongs < 50)){
  System.out.println( “Book for one night” );
} else if ((numOfSongs >= 50)) {
  System.out.println( “Book for two nights” );
} else {
  System.out.println( “Book for the week” );
}
```

66

switch 语句

switch 语句是一个控制流语句，它首先会有一个表达式，然后基于表达式的值将控制流转移至某个 case 语句中。switch 能够与 char、byte、short、int 及其包装类型 Character、Byte、Short 和 Integer、枚举类型以及 String 类型协同使用。对 String 对象的支持是在 Java SE 7 中添加进来的。break 语句能够用来退出 switch 语句。如果 case 语句不包含 break，

在 case 语句完成之后的代码行将会执行。

代码会继续执行，直到遇到 break 语句或者到达 switch 的结尾。这里允许使用 default 标签，为了实现易读性，它通常会列在最后：

```
String style;
String guitarist = “Eric Clapton” ;
...
switch (guitarist) {
  case “Chet Atkins” :
    style = “Nashville sound” ;
    break;
  case “Thomas Emmanuel” :
    style = “Complex fingerstyle” ;
    break;
  default:
    style = “Unknown” ;
    break;
}
```

迭代语句

for 循环、增强的 for 循环、while、do-while 语句是迭代语句，它们用来迭代执行代码片段。

67 for 循环

for 语句包含三部分：初始化、表达式以及更新。如下面的例子所示，变量（比如，i）在使用之前必须要进行初始化。表达式（比如，i<bArray.length）在循环（如，i++）执行之前都会进行调用。只有在表达式的值为 true 的时候，迭代才会进行，在每次迭代之后，变量都会进行更新：

```
Banjo [] bArray = new Banjo[2];
bArray[0] = new Banjo();
bArray[0].setManufacturer( “Windsor” );
bArray[1] = new Banjo();
bArray[1].setManufacturer( “Gibson” );
for (int i=0; i<bArray.length; i++){
  System.out.println(bArray[i].getManufacturer());
}
```

增强的 for 循环

增强的 for 循环，也被称为“for in”循环和“for each”循环，能够用来迭代一个 iterable 的对象或数组。集合或数组中的每个元素都会执行一遍循环，这个过程中不需要使用计数器，因为迭代的次数是已经确定好的：

```
ElectricGuitar eGuitar1 = new ElectricGuitar();
eGuitar1.setName( “Blackie” );
ElectricGuitar eGuitar2 = new ElectricGuitar();
eGuitar2.setName( “Lucille” );
ArrayList <ElectricGuitar> eList = new ArrayList<>();
eList.add(eGuitar1); eList.add(eGuitar2);
for (ElectricGuitar e : eList) {
  System.out.println( “Name:” + e.getName());
}
```

while 循环

68

在 while 语句中，首先会执行表达式，只有当表达式的计算结果为 true 时，才会执行循环。表达式的类型需要是 boolean 或 Boolean：

```
int bandMembers = 5;
while (bandMembers > 3) {
  CoffeeShop c = new CoffeeShop();
  c.performGig(bandMembers);
  // 设置为 0 到 7 之间的任意值
  bandMembers = new Random().nextInt(8);
}
```

do while 循环

在 do while 语句中，循环至少会执行一次，如果表达式的值为 true，那么它会继续执行。表达式的类型需要是 boolean 或 Boolean：

```
int bandMembers = 1;
do {
  CoffeeShop c = new CoffeeShop();
  c.performGig(bandMembers);
  Random generator = new Random();
  bandMembers = generator.nextInt(7) + 1; // 1-7
```

```
} while (bandMembers > 3);
```

控制转移

控制转移语句用来改变程序的控制流。它们包括 break、continue 和 return 语句。

break 语句

未加标签的 break 语句能够用来从 switch 语句中退出或者立即从包含它的循环体中退出。循环体包括 for 循环、增强的 for 循环、while 以及 do-while 迭代语句：

69

```
Song song = new Song( “Pink Panther” );
Guitar guitar = new Guitar();
int measure = 1; int lastMeasure = 10;
while (measure <= lastMeasure) {
  if (guitar.checkForBrokenStrings()) {
    break;
  }
  song.playMeasure(measure);
  measure++;
}
```

添加标签的 break 语句能够强制从所标记的循环中跳出。标签通常会与 for 和 while 组合，用于存在嵌套循环的场景中，此时需要识别要跳出哪个循环。要标记某个循环或语句时，只需将标签放到循环和语句的前面即可，如下所示：

```
...
playMeasures:
while (isWithinOperatingHours()) {
  while (measure <= lastMeasure) {
    if (guitar.checkForBrokenStrings()) {
      break playMeasures;
    }
    song.playMeasure(measure);
    measure++;
  }
} // 退出到这里
```

continue 语句

当执行时，未加标签的 continue 语句会停止当前 for 循环、增强 for 循环、while、do-while 语句的执行，然后试图开始该循环的下一轮迭代执行，接下来会执行循环条件的规则判断。添加标签的 continue 语句找到符合标签的循环语句，然后立即执行下一轮迭代：

```
for (int i=0; i<25; i++) {
  if (playList.get(i).isPlayed()) {
    continue;
  } else {
    song.playAllMeasures();
  }
}
```


return 语句

return 语句用来退出一个方法，如果该方法指定了返回类型，将会返回一个值：

```
private int numberOfFrets = 18; // default
...
public int getNumberOfFrets() {
  return numberOfFrets;
}
```

如果方法不返回任何内容并且到了方法中的最后一条语句，那么 return 语句是可选的。

synchronized 语句

Java 关键字 synchronized 可以用来限制只能有一个线程能够对代码区（如整个方法）进行访问。它能够提供多线程共享资源的访问控制功能。参见第 14 章了解更多信息。

断言语句

断言（assertion）是 Boolean 表达式，能够在 debug 模式（比如在 Java 解释器上使用 -enableassertions 或 -ea 开关）运行时

判断代码的行为是否符合预期。我们可以将断言写成如下所示的形式：

```
assert boolean_expression;
```

断言有助于更容易地识别缺陷，包括识别预期之外的值。它们的设计目标是校验假设始终为 `true`。在 debug 模式运行时，如果断言为 `false`，将会抛出 `java.lang.AssertionError` 并且程序会退出，否则，就像什么事情都没有发生一样：

```
// “strings”的值应该是 4、5、6、7、8 或 12
assert (strings == 12 ||
  (strings >= 4 && strings <= 8));
```

断言功能需要显式启用，参见第 10 章了解启用断言的命令行参数。

断言在编写时还可以包含可选的错误码。尽管将其称为错误（error）码，但它仅仅是一个文本或值，用来达到一些额外信息方面的目的。

当包含错误码的断言执行结果为 `false` 时，错误码的值会转换成一个字符串并且会在程序退出之前将这个值展现给用户：

```
assert boolean_expression : errorcode;
```

使用错误码的一个断言样例如下所示：

```
// 显示非法的“stringed instruments”字符串值
assert (strings == 12 ||
(strings >= 4 && strings <= 8))
: "Invalid string count: " + strings;
```

异常处理语句

异常处理语句用来指定非正常环境下要执行的代码。关键字 `throw` 和 `try/catch/finally` 可以用来进行异常处理。关于异常处理的更多信息，请参见第 7 章。

第 7 章

异常处理

异常是非正常的条件，它会变更或中断执行流。Java 提供了内置的异常处理机制来应对这些条件。异常处理不应该是正常程序流程的一部分。

异常层级结构

如图 7-1 所示，所有的异常和错误都继承自 `Throwable` 类，而 `Throwable` 类继承自 `Object` 类。

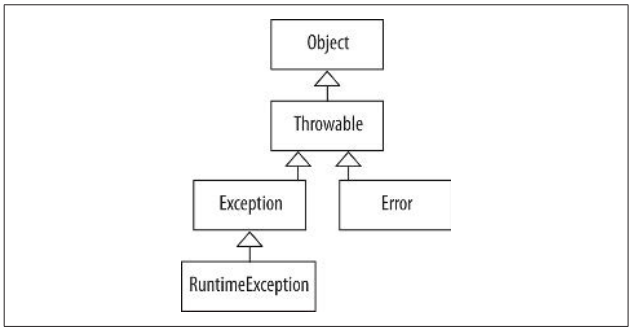


图7.1　异常层级结构的快照

74 检查型 / 非检查型异常和错误

异常和错误分为三类：检查型异常（checked exception）、非检查型异常（unchecked exception）以及错误（error）。

检查型异常

- 检查型异常在编译期会被编译器进行检查；
- 抛出检查型异常的方法必须在方法声明时使用 `throws` 子句（clause）进行声明。这会在调用链上一直向上传播，直到该异常得到处理；
- 所有的检查型异常必须要通过 `catch` 代码块进行显式捕获；
- 检查型异常包括 `Exception` 类型及其子类，但是 `RuntimeException` 以及 `RuntimeException` 的子类不包含在内。

如下是抛出检查型异常的一个方法的样例：

```
// 会抛出 IOException 异常的方法声明
void readFile(String filename)
  throws IOException {
  ...
}
```

非检查型异常

- 非检查型异常在编译器并不会被编译器检查；
- 非检查型异常会在运行时因为程序错误（如索引越界、被零除以及空指针异常）或系统资源耗尽而触发；
- 75 非检查型异常并非必须要被捕获；
- 抛出非检查型异常的方法不必强制（但是可以）在方法声明中标明该异常；
- 非检查型异常包括 `RuntimeException` 类型及其所有子类型的异常。

错误

- 错误一般是不可恢复的，代表了严重的问题；

- 错误在编译期不会进行检查，也不必强制（但是可以）捕获 / 处理。

> **提示**
>
> 所有的检查型异常、非检查型异常以及错误都可以捕获。

常见的检查型 / 非检查型异常和错误

在标准 Java 平台中包含各种检查型异常、非检查型异常以及非检查型错误，其中有些错误和异常比其他的更易于出现。

76

常见的检查型异常

`ClassNotFoundException`

当某个类因为其定义无法找到而导致无法加载时所抛出的异常。

`IOException`

当操作出现故障或中断时抛出的异常。`IOException` 两个常见的子类是 `EOFException` 和 `FileNotFoundException`。

`FileNotFoundException`

当试图打开一个无法找到的文件时，将会抛出该异常。

`SQLException`

当出现数据库错误时，将会抛出该异常。

`InterruptedException`

当线程被中断的时候，将会抛出该异常。

`NoSuchMethodException`

当调用无法找到的方法时，将会抛出该异常。

`CloneNotSupportedException`

当不支持克隆的对象调用 clone() 时将会抛出该异常。

常见的非检查型异常

ArithmeticException

抛出该异常表明出现了意料之外的算术条件。

ArrayIndexOutOfBoundsException

抛出该异常表明索引越界。

ClassCastException

当某个对象转换为一个子类，但该对象并不是该子类的实例时，将会抛出该异常。

DateTimeException

抛出该异常表明出现了日期 - 时间对象创建、查询和操作错误。

IllegalArgumentException

抛出该异常表明将非法的参数传递给了方法。

77 IllegalStateException

抛出该异常表明在不恰当的时间调用了某个方法。

IndexOutOfBoundsException

抛出该异常表明索引越界。

NullPointerException

当代码引用了 null 对象，但需要一个非空对象时，将会抛出该异常。

NumberFormatException

抛出该异常表明非法地将字符串转换为数字类型。

UncheckedIOException

使用非检查异常包装 IOException。

常见的错误

`AssertionError`

抛出该错误表明出现了断言错误。

`ExceptionInInitializeError`

抛出该错误表明在静态初始化器中出现了非检查异常。

`VirtualMachineError`

抛出该错误表明 JVM 出现了问题。

`OutOfMemoryError`

当没有足够的内存存放对象或执行垃圾收集时，将会抛出该错误。

`NoClassDefFoundError`

当某个类在编译时能够找到，但是运行时 JVM 无法找到的话，将会抛出该错误。

`StackOverflowError`

抛出该错误表明出现了栈溢出。

异常处理的关键字

78

在 Java 中，处理错误的代码是与生成错误的代码清晰分离开的。生成异常的代码被称为“抛出”异常，而处理异常的代码被称为“捕获”异常：

```
// 声明一个异常
public void methodA() throws IOException {
  ...
  throw new IOException();
  ...
}

// 捕获异常
public void methodB() {
  ...
  /* 对 methodA 的调用必须放到一个 try/catch 代码块中，因为这个异常是检查型异常，如果不这样做，methodB 也需要抛出该异常 */
  try {
```

```
        methodA();

      }catch (IOException ioe) {
        System.err.println(ioe.getMessage());
        ioe.printStackTrace();
      }
    }
```

throw 关键字

抛出一个异常时，要使用 throw 关键字。所有的检查型 / 非检查型异常以及错误都可以抛出：

```
if (n == -1)
  throw new EOFException();
```

try/catch/finally 关键字

抛出的异常会被 Java 的 try、catch、finally 代码块所处理。Java 解释器会寻找处理异常的代码，首先会在封闭的代码块中进行查找，必要的话，将会沿着调用栈向上传播至 main() 方
79 法。如果异常没有在主线程（也就是，非事件分发线程，Event Dispatch Tread[EDT]）或你所创建的线程中进行处理，那么程序将会退出并打印栈跟踪信息：

```
try  {
  method();
} catch (EOFException eofe) {
  eofe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
} finally {
  // cleanup
}
```

try-catch 语句

try-catch 语句包含一个 try 以及一个或多个 catch 代码块。

try 代码块中包含了可能会抛出异常的代码。所有可能抛出的检查型异常必须要有一个 catch 代码块来处理该异常。如果没有异常抛出，try 代码块会正常终止。一个 try 代码块会有零

个或多个 catch 子句来处理异常。

提示

try 代码块必须至少有一个 catch 或 finally 代码块与之关联，它也可以忽略异常，只需在方法上将异常抛出即可。

在 try 代码块与 catch 代码块（如果存在）或 finally 代码块（如果存在）之间，不能有任何的代码。

catch 代码块包含了处理所抛出异常的代码，包括将异常信息打印到文件中，这样的话，能够提示用户输入正确的信息。需要注意的是，catch 代码块不应该是空的，因为如果这样做，这种“沉默”的结果会将异常隐藏起来，让错误更加难以调试。

80

在 catch 子句中命名参数时，一种常见的约定是采用能够代表异常的每个单词的一组字母：

```
catch (ArrayIndexOutOfBoundsException aioobe) {
  aioobe.printStackStrace();
}
```

在 catch 子句中，如果必要，可能会抛出新的异常。

在 try/catch 代码中，try/catch 子句的顺序定义了异常捕获的优先级。我们始终要从更具体的异常开始，最后才是可能抛出的最通用的异常。

提示

try 代码块抛出的异常会直接定向到第一个与该异常类型相同或者是该异常超类的 catch 子句中。具有 Exception 参数的 catch 代码块应该始终放到顺序列表的最后一项中。

如果没有任何 catch 子句的参数匹配所抛出的异常，系统将会查找匹配该异常超类的参数。

try-finally 语句

try-finally 语句包含一个 try 和一个 finally 代码块。如果必要，可以使用 finally 代码块来释放资源：

81
```
public void testMethod() throws IOException {
FileWriter fileWriter =
  new FileWriter( “\\data.txt” );
  try {
    fileWriter.write( “Information...” );
  } finally {
    fileWriter.close();
  }
}
```

这个代码块是可选的，只有在需要的地方才会使用。如果使用，它会在 try-finally 代码块的最后执行，并且不管 try 代码块是否正常终止，它都会始终执行。如果 finally 代码块抛出异常，它必须要进行处理。

try-catch-finally 语句

try-catch-finally 语句包括一个 try、一个或多个 catch 代码块以及一个 finally 代码块。

在这个语句中，finally 代码块用来清理和释放资源：

```
public void testMethod() {
  FileWriter fileWriter = null;
  try {
    fileWriter = new FileWriter( “\\data.txt” );
    fileWriter.write( “Information...” );
  } catch (IOException ex) {
    ex.printStackTrace();
  } finally {
    try {
      fileWriter.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

```
}
```

finally代码块是可选的，只有在需要的地方才会使用。如果使用，它会在 try-catch-finally 代码块的最后执行，并且不管 try 代码块是否正常终止或者 catch 子句是否执行过，它都会始终执行。如果 finally 代码块抛出异常，它必须要进行处理。

82

try-with-resources 语句

try-with-resources 用来声明资源必须要在不使用的时候进行关闭。这些资源是在 try 中声明的。Java 9 简化了该语句：

```
// Java 7 and 8
public void testMethod() throws IOException {
  FileWriter fileWriter = new FileWriter( "\\data.
txt" );
  try (FileWriter fw = fileWriter)
  {
     fw.write( "Information..." );
  }
}
// Java 9
public void testMethod() throws IOException {
  FileWriter fileWriter = new FileWriter( "\\data.
txt" );
  try (fileWriter)
  {
     f1.write( "Information..." );
  }
}
```

任何实现了 AutoClosable 接口的资源都可以与 try-with-resources 语句协同使用。

多 catch 子句

多 catch 子句能够用来在一个 catch 子句接受多个异常参数：

```
boolean isTest = false;
public void testMethod() {
  try {
    if (isTest) {
      throw new IOException();
    } else {
      throw new SQLException();
```

83

```
        }
      } catch (IOException | SQLException e) {
        e.printStackTrace();
      }
    }
```

异常处理的过程

下面列出了异常处理流程的几个步骤：

1. 当遇到异常时，将会创建一个异常对象；
2. 抛出新的异常对象；
3. 运行时系统寻找代码来处理异常，首先从异常对象创建的方法开始。如果找不到处理器，运行时环境将会反向遍历调用栈（方法的有序列表）来查找异常处理器。如果异常依然得不到处理，那么程序将会退出并自动打印栈跟踪信息；
4. 运行时系统会将异常对象交给异常处理器来处理（捕获）异常。

定义自己的异常类

除了已有的 Java 异常之外，如果必要，编程人员可以定义自己的异常。通常来讲，如果可能，Java 异常应该进行重用：

- 要定义检查型异常，新的异常类必须直接或间接扩展 `Exception` 类；
- 要定义非检查型异常，新的异常类必须直接或间接扩展 `RuntimeException` 类；
- 要定义非检查型错误，新的错误类必须扩展 `Error` 类。

84

用户定义的异常至少要有两个构造器，一个是不接受任何参数的构造器，另外一个构造器如下所示：

```
public class ReportException extends Exception {
  public ReportException () {}
  public ReportException (String message, int
    reportId) {
    ...
  }
```

```
}
```

如果需要捕获某个异常并抛出更为具体的异常，那么较为明智的做法是捕获基础异常。

打印异常信息

在 Throwable 类中，提供所抛出异常信息的方法是 getMessage()、toString() 和 printStackTrace()。通常而言，在 catch 子句中处理异常时，需要调用其中的某个方法。编程人员也可以编写代码在异常发生时获取额外的有用信息（比如，无法找到的文件名称是什么）。

getMessage() 方法

getMessage() 方法应当以字符串形式返回异常的详细信息：

```
try {
  new FileReader( “file.js” );
} catch (FileNotFoundException fnfe) {
  System.err.println(fnfe.getMessage());
}
```

85

toString() 方法

toString () 方法以字符串形式返回异常的详细信息，其中还包括它的类名：

```
try {
  new FileReader( “file.js” );
} catch (FileNotFoundException fnfe) {
    System.err.println(fnfe.toString());
}
```

printStackTrace() 方法

printStackTrace() 方法以字符串形式返回异常的详细信息，其中包括它的类名以及从错误捕获开始的栈跟踪信息，一直回溯到异常抛出的地方为止：

```
try {
  new FileReader( “file.js” );
} catch (FileNotFoundException fnfe) {
  fnfe.printStackTrace();
}
```

下面展现了栈跟踪信息的一个样例。第一行包含了异常对象的 `toString()` 方法调用时所返回的内容。剩下的部分展现了方法的调用，从异常抛出的地方开始，一路回溯到它抛出和被处理的地方：

```
java.io.FileNotFoundException: file.js (The system
cannot find the file specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.(init)
 (FileInputSteam.java:106)
 at java.io.FileInputStream.(init)
 (FileInputSteam.java:66)
 at java.io.FileReader(init)(FileReader.java:41)
 at EHExample.openFile(EHExample.java:24)
 at EHExample.main(EHExample.java:15)
```

86

提示

Java 9 引入了 Stack-Walking API，允许我们更容易地过滤和懒加载栈跟踪中的信息。

第 8 章

Java修饰符

修饰符是 Java 关键字，可以用到类、接口、构造器、方法和数据成员上。

表 8-1 列出了 Java 修饰符及其适用的场景。需要注意允许使用私有和受保护的类，但是只能作为内部类或嵌套类。参考第 21 章了解 Java 9 相关的可访问性细节。

表8-1　Java修饰符

修饰符	类	接口	构造器	方法	数据成员
访问修饰符					
包私有	是	是	是	是	是
`private`	否	否	是	是	是
`protected`	否	否	是	是	是
`public`	是	是	是	是	是
其他修饰符					
`abstract`	是	是	否	是	否
`final`	是	否	否	是	是
`native`	否	否	否	是	否
`strictfp`	是	是	否	是	否
`static`	否	否	否	是	是
`synchronized`	否	否	否	是	否

修饰符	类	接口	构造器	方法	数据成员
transient	否	否	否	否	是
volatile	否	否	否	否	是

内部类有可能会使用 private 或 protected 访问修饰符。本地变量只可能使用一个修饰符：final。

访问修饰符

访问修饰符定义了类、接口、构造器、方法和数据成员的访问权限。访问修饰符包括 public、private 和 protected。如果没有指定修饰符，将会使用包私有的访问权限。

表 8-2 描述了使用访问修饰符时所对应的可见性细节信息。

表8-2　访问修饰符及其可见性

修饰符	可见性
包私有	默认的包私有限制只能在包中进行访问
private	private 方法只能在它所在的类中访问。private 数据成员只能在它所在的类或接口（Java 9）中访问。它能够通过方法进行间接访问（如 getter 和 setter 方法）
protected	protected 方法能够在它所在的包中进行访问，而且能够被位于包之外且包含该方法的类的子类所访问； protected 数据成员能够在它的包中进行访问，而且能够被位于包之外且包含该数据成员的类的子类所访问
public	public 修饰符允许在任何地方访问，即便是在声明它的包之外也可以访问。注意，接口默认就是 public 的。

其他（非访问）修饰符

表 8-3 包含了非访问 Java 修饰符及其用途。

表8-3　非访问Java修饰符

修饰符	用途
abstract	抽象类是使用关键字 abstract 声明的类。它不能同时使用 final 来声明。接口默认就是抽象的，不必使用 abstract 来进行声明。
	abstract 方法只包含签名而没有方法体。如果类中有一个方法是抽象的，那其所在的类就要是抽象的。抽象方法不能与 final、native、private、static 或 synchronized 一起使用
default	default 方法，也被称为守护者（defender），允许在接口中创建默认的方法实现
final	final 类不能进行扩展。
	final 方法不能覆盖。
	final 数据成员只能初始化一次不能进行变更。声明为 static final 的数据成员会在编译时进行设置，并且无法修改
native	native 方法用来合并其他编程语言（比如 C 和 C++）的代码到 Java 程序中。它只包含一个签名而没有方法体。它不能与 strictfp 协同使用
static	static 方法和 static 变量能够通过类名进行访问。它们能够在整个类和该类所有实例上使用。
	static 数据成员能够通过类名访问。不管这个类有多少实例，static 的数据成员只存在一份

90

修饰符	用途
strictfp	strictfp 类会按照 IEEE 754-1985 浮点规范来进行所有的浮点操作。 strictfp 方法具有 FP-strict 方法中所有的表达式。接口中的方法不能声明为 strictfp，它不能与 native 修饰符一同使用
synchronized	synchronized 关键字保证在同一个时间内只有一个线程能够执行方法中的代码块，使其保持线程安全。语句也可以使用 synchronized
transient	transient 数据成员在整个类序列化时，它不会序列化。它不是类持久化状态的一部分
volatile	volatile 数据成员会通知线程，要获取该变量的最新值（而不是使用缓存副本）并且要在变量更新时立即写回最新的值

修饰符的编码

应用到类和成员上的修饰符可以分别通过 Class.getModifiers() 和 Member.getModifiers() 获取到。修饰符是一个编码过的值，可以通过 Modifier.toString(value) 进行解码：

```
// HashMap 类上所用的修饰符
Class c = new HashMap().getClass();
String modifiers = Modifier.toString(c.getModifiers());
System.out.println( “Class Modifier(s) = “ + modifiers);
$ Class Modifier(s) = public
```

91

```
// Hashmap isEmpty 成员 / 方法上所用的修饰符
Member m = new HashMap().getClass().
getDeclaredMethod( “isEmpty” );
String modifiers = Modifier.toString(m.getModifiers());
System.out.println( “Method Modifier(s) = “ +
modifiers);
$ Method Modifier(s) = public
```

第 2 部分

平台

第 9 章

Java平台，标准版

Java 平台，标准版（Standard Edition，SE），包括了 Java 运行时环境（Java Runtime Environment，JRE）以及与其兼容的 Java 开发工具集（Java Development Kit，JDK，参见第 10 章）、Java 编程语言、Java 虚拟机（Java Virtual Machine，JVM）、工具集（tools/utilities）和 Java SE API 库。它支持各种平台，包括 Windows、macOS、Linux 和 Solaris。

常用的 Java SE API 库

Java SE API 标准库（*https://docs.oracle.com/javase/9/docs/api/*）是在包（和模块）中提供的。每个包都由类和 / 或接口组成。我们在这里给出了常用包的列表以阐述 API 的功能。这里的列表是与 JDK 9 中的模块顺序无关的。

Java SE 从 Java SE 7 update 6 版本开始提供了 JavaFX 运行时库，并演化到了 JavaFX 2.2（*https://docs.oracle.com/javafx/2/api/*）。JavaFX 已经取代了 Swing API，成为了 Java SE 中首选的客户端 UI 库。

语言和工具库

`java.lang`

语言支持：系统 / 数学方法、基本类型、字符串、线程和异常。

96 `java.lang.annotation`

注解框架：支持元数据的库。

`java.lang.instrument`

程序 instrumentation：instrument JVM 程序的代理服务。

`java.lang.invoke`

动态语言支持：由核心类和 VM 提供支持。

`java.lang.management`

Java 管理扩展（Java Management Extension）；JVM 监控和管理。

`java.lang.module`

模块描述符（descriptor）和配置支持。

`java.lang.ref`

引用对象类：与 GC 进行交互的支持功能。

`java.lang.reflect`

关于类和对象的反射信息。

`java.util`

工具集：对集合、事件模型、日期 / 时间以及国际化的支持。

`java.util.concurrent`

并发工具集：执行器（executor）、队列、计时以及同步。

`java.util.concurrent.atomic`

原子工具集：针对单个变量提供无锁且线程安全的编程方式。

`java.util.function`

函数式接口：为 lambda 表达式和方法引用提供了目标类型。

`java.util.jar`

Java 归档（Java Archive）文件格式：读入和写入。

`java.util.logging`

日志：故障、错误、性能问题以及缺陷的记录。

`java.util.prefs`

用户和系统的偏好设置（preference）：检索和存储。

`java.util.regex`

正则表达式：匹配模式的字符序列。

`java.util.stream`

流：对元素流的函数式操作。

`java.util.zip`

ZIP 和 GZIP 文件格式：读取和写入。

基础库

`java.beans`

bean：基于 JavaBean、长期持久化的组件。

`java.beans.beancontext`

bean 上下文：bean 容器，运行时环境。

`java.io`

输入 / 输出：数据流、文件系统以及序列化。

`java.math`

数学：特别大的整数和小数的运算。

`java.net`

网络：TCP、UDP、套接字和地址。

`java.nio`

高性能 I/O：缓冲（buffer）以及内存映射文件（memory-mapped file）。

`java.nio.channels`

I/O 的通道（channel）：非阻塞 I/O 的选择器（selector）。

98 `java.nio.charset`

字符集：在字节和 Unicode 之间进行转换。

`java.nio.file`

文件的支持：文件、文件属性以及文件系统。

`java.nio.file.attribute`

文件和文件系统属性的支持。

`java.text`

文本工具：文本、日期、数字和消息。

`java.time`

时间：日期、时间、时刻和持续时间。

`java.time.chrono`

时间：日历系统。

`java.time.format`

时间：打印和解析。

`java.time.temporal`

时间：通过字段、单位和调整器（adjuster）进行访问。

`java.time.zone`

时间：支持时区及其规则。

`javax.annotation`

注解类型：库方面的支持。

`javax.management`

JMX API：应用配置、统计以及状态变化。

`javax.net`

网络：套接字工厂。

`javax.net.http`

高层级的 HTTP 和 WebSocket API。

`javax.net.ssl`

安全套接字层：错误探测、数据加密 / 认证。

`javax.tools`

程序调用工具接口：编译器、文件管理。

99

集成库

`java.sql`

结构化查询语言（Structured Query Language，SQL）：访问和处理数据源信息。

`javax.jws`

Java Web 服务：支持注解类型。

`javax.jws.soap`

Java Web 服务：SOAP 绑定和消息参数。

`javax.naming`

命名服务：Java 命名和目录接口（Java Naming and Directory Interface，JNDI）。

`javax.naming.directory`

目录服务：针对目录存储对象的 JNDI 操作。

`javax.naming.event`

事件服务：JNDI 事件通知操作。

`javax.naming.ldap`

轻量级目录访问协议 v3（Lightweight Directory Access Protocol v3）：操作与控制。

`javax.script`

脚本语言支持：集成、绑定与调用。

`javax.sql`

SQL：数据库 API 以及服务端的功能。

`javax.sql.rowset.serial`

序列化映射：SQL 类型和数据类型之间的映射。

`javax.sql.rowset`

Java 数据库连接（Java Database Connectivity，JDBC）Rowset：标准接口。

100 `javax.transactions.xa`

XA 接口：针对 JTA 的事务和资源管理协议。

各种用户界面相关的库

`javax.accessibility`

可访问性技术：UI 组件的辅助支持。

`javax.imageio`

Java 图片 I/O：图片文件内容描述（元数据、缩略图）。

`javax.print`

打印服务：格式化和任务提交。

`javax.print.attribute`

Java 打印服务：属性和属性集收集。

`javax.print.attribute.standard`

标准属性：广泛使用的属性和值。

`javax.print.event`

打印事件：服务和打印任务监控。

`javax.sound.midi`

声音：I/O、排序以及 MIDI 类型 0 和 1 的合成。

`javax.sound.sampled`

声音：音频数据采样 (AIFF、AU 和 WAV 格式)。

JavaFX 用户界面库

`javafx.animation`
基于转换（transition）的动画。

`javafx.application`
应用生命周期。

`javafx.beans`
可观测性（observability）的通用形式。

`javafx.beans.binding`
绑定特征。

101

`javafx.beans.property`
只读和可写的属性。

`javafx.beans.property.adapter`
属性适配器。

`javafx.beans.value`
读取和写入。

`javafx.collections`
JavaFX 集合工具集。

`javafx.concurrent`
JavaFX 并发任务。

`javafx.embed.swing`
Swing API 应用的集成。

`javafx.embed.swt`
SWT API 应用的集成。

`javafx.event`
事件框架（如投递和处理）。

`javafx.fxml`

标记语言（如加载对象层级结构）。

`javafx.geometry`

二维几何。

`javafx.scene`

基础类：核心的 Scene Graph API。

`javafx.scene.canvas`

Canvas 类：一种即时模式的渲染 API。

`javafx.scene.chart`

图表组件：数据可视化。

`javafx.scene.control`

用户界面控件：场景图中的特定节点。

102 `javafx.scene.control.cell`

单元格相关的类（也就是非核心类）。

`javafx.scene.effect`

图形化过滤效果：支持场景图节点。

`javafx.scene.image`

加载和展现图片。

`javafx.scene.input`

鼠标和键盘输入事件的处理。

`javafx.scene.layout`

界面布局的类。

`javafx.scene.media`

音频和视频的类。

`javafx.scene.paint`

颜色和渐变的支持 (例如，填充形状和背景)。

`javafx.scene.shape`

二维形状。

`javafx.scene.text`

字体和文本节点的渲染。

`javafx.scene.transform`

转换：仿射对象（affine object）的旋转、缩放、剪切和转换。

`javafx.scene.web`

Web 内容：加载和展现 Web 内容。

`javafx.stage`

Stage：顶级的容器。

`javafx.util`

工具和辅助类。

`javafx.util.converter`

字符串转换。

远程方法调用（RMI）和 CORBA 库

103

`java.rmi`

远程方法调用（Remote Method Invocation）：调用远程 JVM 中的对象。

`java.rmi.activation`

RMI 对象的激活：激活持久化远程对象的引用。

`java.rmi.dgc`

RMI 分布式垃圾收集（distributed garbage collection，DGC）：从客户端进行远程对象跟踪。

`java.rmi.registry`

RMI 注册表：将名字与远程对象进行匹配。

`java.rmi.server`

RMI 服务端：RMI 传输协议，超文本传输协议（Hypertext

Transfer Protocol，HTTP）隧道、存根（stub）。

`javax.rmi`

远程方法调用（RMI）：远程方法调用 Internet InterORB 协议（Remote Method Invocation Internet InterORB Protocol，RMI-IIOP）、Java 远程方法协议（Java Remote Method Protocol，JRMP）。

`javax.rmi.CORBA`

公共对象请求代理体系结构（Common Object Request Broker Architecture，CORBA）：针对 RMI-IIOP 和对象请求代理（Object Request Broker，ORB）的可移植 API。

`javax.rmi.ssl`

安全套接字层（Secured Sockets Layer，SSL）：RMI 客户端和服务器的支持。

`org.omg.CORBA`

OMG CORBA ：CORBA 到 Java 的映射，ORB。

`org.omg.CORBA_2_3`

104

OMG CORBA 的附加功能：进一步的 Java 兼容性套件（Java Compatibility Kit，JCK）测试支持。

安全库

`java.security`

安全：算法、机制和协议。

`java.security.cert`

证书：解析、管理证书撤销列表（Certificate Revocation List，CRL）和证书路径。

`java.security.interfaces`

安全接口：Rivest、Shamir 与 Adelman (RSA) 和 Digital Signature Algorithm (DSA) 的生成。

`java.security.spec`

规范：安全密钥和算法参数。

`javax.crypto`

密码操作：加密、密钥和 MAC 生成。

`javax.crypto.interfaces`

Diffie-Hellman 密钥：在 RSA 实验室的 PKCS #3 中定义。

`javax.crypto.spec`

规范：针对安全密钥和算法参数。

`javax.security.auth`

安全认证和授权：访问控制规范。

`javax.security.auth.callback`

认证回调的支持：与应用的服务交互。

`javax.security.auth.kerberos`

Kerberos 网络认证协议：相关的工具类。

`javax.security.auth.login`

登录和配置：可插拔的认证框架。

`javax.security.auth.x500`

X500 主体（Principal）和 X500 私有凭证（Private Credential）：主题存储。

`javax.security.sasl`

简单认证和安全层（Simple Authentication and Security Layer，SASL）：SASL 认证。

`org.ietf.jgss`

Java 通用安全服务（Java Generic Security Service，JGSS）：认证、数据完整性。

可扩展标记语言（XML）库

`javax.xml`

扩展性标记语言（XML）：常量。

`javax.xml.bind`

XML 运行时绑定：解排（unmarshalling）、编排（marshalling）和校验。

`javax.xml.catalog`

XML 目录支持：XML Catalogs OASIS Standard V1.1，2005 年 10 月 7 日。

`javax.xml.crypto`

XML 加密：签名生成和数据加密。

`javax.xml.crypto.dom`

XML 和文档对象模型（Document Object Model，DOM）：加密实现。

`javax.xml.crypto.dsig`

XML 数字签名：生成和校验。

`javax.xml.datatype`

XML 和 Java 数据类型：映射。

106 `javax.xml.namespace`

XML 命名空间：处理。

`javax.xml.parsers`

XML 解析：Simple API for XML（SAX）和 DOM 解析器。

`javax.xml.soap`

XML SOAP 消息：创建和构建。

`javax.xml.transform`

XML 转换处理：不依赖 DOM 或 SAX。

`javax.xml.transform.dom`

XML 转换处理：DOM 特定的 API。

`javax.xml.transform.sax`

XML 转换处理：SAX 特定的 API。

`javax.xml.transform.stax`

XML 转换处理：Streaming API for XML (StAX) API。

`javax.xml.validation`

XML 校验：基于 XML 模式（schema）进行验证。

`javax.xml.ws`

针对 XML Web Services 的 Java API（JAX-WS）：核心 API。

`javax.xml.ws.handler`

JAX-WS 消息处理器：消息上下文和处理器接口。

`javax.xml.ws.handler.soap`

JAX-WS：SOAP 消息处理器。

`javax.xml.ws.http`

JAX-WS：HTTP 绑定。

`javax.xml.ws.soap`

JAX-WS：SOAP 绑定。

`javax.xml.xpath`

XPath 表达式：执行和访问。

107

`org.w3c.dom`

W3C 的 DOM：访问和更新文件内容与结构。

`org.xml.sax`

XML.org 的 SAX：访问和更新文件内容与结构。

第 10 章

开发的基础工具 109

Java 运行时环境（Java Runtime Environment，JRE）提供了运行 Java 应用的支柱。Java 开发工具集（Java Development Kit，JDK）提供了开发 Java 应用所需的所有组件和必要的资源。

Java 运行时环境

JRE 是允许计算机系统运行 Java 应用的一组软件。这个软件集包括将 Java 字节码解释为机器码的 Java 虚拟机（Java Virtual Machine，JVM）、标准类库、用户接口工具集以及各种工具。

Java 开发工具集

JDK 是一个编程环境，用来编译、调试和运行 Java 应用和 Java Beans。JDK 包含了 JRE，另外还添加了 Java 编程语言和额外的开发工具与工具 API。Oracle 的 JDK 支持 macOS、Solaris、Linux（Oracle、Suse、Red Hat、Ubuntu 和 Debian [在 ARM 上]）以及 Microsoft Windows（Server 2008 R2、Server 2012、Vista、Windows 7、Windows 8 和 Windows 10）。其他的操作系统和特定目的的 JVM、JDK 和 JRE 都可以从 Java 虚拟机（*http://java-virtual-machine.net/other.html*）站点免费获取。 110

表 10-1 列出了 Oracle 所提供的 JDK 版本。读者可以在 Oracle 的 Web 站点（*http://www.oracle.com/technetwork/java/javase/downloads/index.html*）下载最新的版本，也可以下载较旧的版本（*http://www.oracle.com/technetwork/java/archive-139210.html*）。

表10-1　Java开发工具集

Java 开发工具集	编码名	发布时间	包的数量	类的数量
Java SE 9 with JDK 1.9.0	---	2017	~225	~4,413
Java SE 8 with JDK 1.8.0	---	2014	217	4,240
Java SE 7 with JDK 1.7.0	Dolphin	2011	209	4,024
Java SE 6 with JDK 1.6.0	Mustang	2006	203	3,793
Java 2 SE 5.0 with JDK 1.5.0	Tiger	2004	166	3,279
Java 2 SE with SDK 1.4.0	Merlin	2002	135	2,991
Java 2 SE with SDK 1.3	Kestrel	2000	76	1,842
Java 2 with SDK 1.2	Playground	1998	59	1,520
Development Kit 1.1	---	1997	23	504
Development Kit 1.0	Oak	1996	8	212

2015 年 4 月，Oracle 结束了 Java SE 7 版本的公开更新。

Java 程序结构

Java 的源文件可以通过 jEdit、TextPad、Vim、Notepad++ 这样的文本编辑器创建，也可以通过 Java 集成开发环境（Integrated Development Environment，IDE）来提供。源码文件必须以“.java”作为扩展名，并且名称要与文件中公开类的名称相同。如果类是包私有的，那么类名可以与文件名不相同。

因此，名为 *HelloWorld.java* 的源文件要与名为 HelloWorld 的公开类相对应，如下面的样例所示（Java 中所有的命名都是区分大小写的）：

111

```
 1 package com.oreilly.tutorial;
 2 import java.time.*;
 3 // import java.time.ZoneId;;
 4 // import java.time.Clock;
 5
 6 public class HelloWorld
 7 {
 8   public static void main(String[] args)
 9   {
10     ZoneId zi = ZoneId.systemDefault();
11     Clock c = Clock.system(zi);
12     System.out.print( “From: “
13       + c.getZone().getId());
13     System.out.println( “, \” Hello, World!\”” );
14   }
15 }
```

在第一行中，声明类 HelloWorld 位于 com.oreilly.tutorial 包中。这个包名意味着 com/oreilly/tutorial 是一个目录结构，并且编译器和运行时环境都能在类路径中访问到它。将源码文件进行分包组织是可选的，但是推荐这样做，因为这样能够避免与其他软件包的冲突。

在第 2 行中，import 的声明允许 JVM 去其他的包中查找类。在这里，使用星号意味着 java.time 包中的所有类都是可用的。但是，你应该尽量显式导入类，这样的话依赖能够记录下来，在这里应该包括 import java.time. ZoneId 和 import java.time.Clock。可以看到，这两句被注释掉了，但是它们应该是比使用 import

java.time.* 更好的方案。需要注意的是，导入语句并不是必需的，因为我们可以在每个类的前面使用完整的包名，不过这并不是编码的理想方式。

112

提示

java.lang 包是唯一默认导入的包。

在第 6 行中，源码文件必须有唯一一个顶级的 `public` 类。除此之外，文件中可以包含多个包私有的类。

在第 8 行中，我们需要注意 Java 应用必须要有一个 `main` 方法。这个方法是 Java 程序的入口，必须要进行定义，其修饰符必须声明为 `public`、`static` 和 `void`。该方法的传入参数提供了一个命令行参数所组成的数组。

提示

容器管理的组件（如 Spring 和 Java EE）并不依赖于 `main` 方法。

在第 12 和第 13 行，表达式调用了 `System.out.print` 和 `System.out.println` 方法，将提供的文本打印到控制台窗口中。

命令行工具

JDK 提供了多个命令行工具辅助软件开发。常用的工具包括编译器、启动器 / 解释器（launcher/interpreter）、归档器（archiver）以及文档器（documenter）。读者可以参见 Oracle 的网站（*https://docs.oracle.com/javase/7/docs/technotes/tools/*）查看完整的工具列表。

Java 编译器

Java 编译器将 Java 源码文件翻译为 Java 字节码。编译器将会创建与源码文件名称相同的字节码文件，不过扩展名为“.class”。113
如下列出了常用的编译器选项：

`javac [-options] [source files]`

编译 Java 源文件。

`javac HelloWorld.java`

编译程序生成 HelloWorld.class。

`javac –cp /dir/Classes/ HelloWorld.java`

参数 `-cp` 和 `–classpath` 是等价的，用来指定编译期的类路径目录。

`javac –d /opt/hwapp/classes HelloWorld.java`

参数 `-d` 用来将生成的类文件放到预先存在的特定目录下。如果存在包定义，路径会包含进来（如 */opt/hwapp/classes/com/oreilly/tutorial/*）。

`javac –s /opt/hwapp/src HelloWorld.java`

参数 -s 将生成的源文件放到预先存在的特定目录下。如果存在包定义，路径会进一步的扩展（如 */opt/hwapp/src/com/oreilly/tutorial/*）。

`javac –-release 8 HelloWorld.java`

参数 `-release` 会基于特定的 VM 版本编译，支持 Java 6、7、8 和 9。

`javac –source 1.8 HelloWorld.java`

参数 `–source` 提供了与给定释放版本的源码兼容性，允许将不支持的关键字用作标识符。

`javac –X`

参数 `–X` 打印非标准用法的诊断信息。例如，`–Xlint:unchecked` 将会启用推荐警告，它会打印出非检查或不安全

操作的详细信息。

114

提示

尽管 –Xlint 和其他的 -X 参数在 Java 编译器中很常见，但是 -X 并没有标准化，所以不能假定它们都能跨 JDK 适用。

`javac –version`

参数 `-version` 打印 `javac` 工具的版本。

`javac –help`

使用 `-help` 参数或者不使用任何参数时，将会打印出 `javac` 命令的帮助信息。

读者请参见第 21 章了解 `javac` 扩展模块功能的额外参数。

Java 解释器

Java 解释器负责程序的执行，包括启动应用。如下是常用的解释器参数：

`java [-options] class [arguments...]`

运行解释器。

`java [-options] –jar jarfile [arguments...]`

执行 JAR 文件。

`java HelloWorld`

启动 JRE，加载 HelloWorld 并运行该类的 `main` 方法。

`java com.oreilly.tutorial.HelloWorld`

启动 JRE，加载 *com/oreilly/tutorial/* 目录下的 HelloWorld 类并运行该类中的 `main` 方法。

`java [-cp | -classpath] /tmp/Classes HelloWorld`

115

参数 `-cp` 和 `-classpath` 指定在运行时所使用的类路径目录。

`java –Dsun.java2d.ddscale=true HelloWorld`

参数 `-D` 设置系统属性的值。在这里将启用硬件加速扩展（hardware accelerator scaling）。

`java [–ea | enableassertions] HelloWorld`

参数 `-ea` 和 `-enableassertions` 用来启用 Java 断言。断言是插入到应用中的诊断代码。关于断言的更多信息，请参见第 6 章“断言语句”一节内容。

`java [-da | disableassertions] HelloWorld`

参数 `–da` 和 `–disableassertions` 用来禁用 Java 断言。

`java –client HelloWorld`

参数 `–client` 会使用客户端虚拟机，它会增强交互式应用的体验，比如 GUI。

`java –server HelloWorld`

参数 `-server` 会使用服务器虚拟机，增强系统的整体性能。

`java –splash:images/world.gif HelloWorld`

借助参数 `-splash` 能够在运行应用之前展现一个图片形式的启动页面。

`java –version`

参数 `–version` 会打印 Java 解释器、JRE 和虚拟机的版本。

`java –help`

使用参数 `-help` 或者不使用任何的参数，将会打印 java 命令的帮助信息。

`javaw <classname>`

在 Windows 操作系统上，`javaw` 等价于 `java` 命令，但是前者没有控制台窗口。在 Linux 上与之等价的运行方式是通过使用 & 符号，将 java 运行为后台进程：`java <classname> &`。 116

Java 程序打包器

Java 归档（Java Archive，JAR）是一个归档和压缩工具，通常用来将多个文件合并到一个 JAR 文件中。JAR 是一个 ZIP 归档，包含了一个清单文件（JAR 内容的描述）和可选的签名文件（为了安全性）。如下是常用的 JAR 参数及其样例：

```
jar [options] [jar-file] [manifest-files] [entry-point] [-C dir] files...
```

JAR 工具的用法：

```
jar cf files.jar HelloWorld.java com/oreilly/tutorial/HelloWorld.class
```

参数 c 允许创建 JAR 文件，f 参数允许指定文件名。在本例中，HelloWorld.java 和 com/oreilly/tutorial/HelloWorld.class 将会打包到 JAR 文件中。

```
jar tfv files.jar
```

参数 t 用来列出 JAR 文件中的内容，f 参数用来指定文件名，v 参数指定以详细格式（verbose format）列出内容。

```
jar xf files.jar
```

参数 x 允许抽取 JAR 文件中的内容，参数 f 用来指定文件名称。

提示

有多个 ZIP 工具（比如 7-Zip、WinZip 和 WinRAR）可以与 JAR 文件协同使用。

117 JAR 文件执行

我们创建了 JAR 文件，接下来通过指定 JAR 中“main”类位于哪个文件中就可以执行了，这样的话，Java 解释器就知道要使用哪个 main() 方法了。如下是执行 JAR 文件的完整样例：

1．通过本章开始时介绍的 HelloWorld 类创建 HelloWorld.java 文件。
2．创建子文件夹 *com/oreilly/tutorial/*。
3．运行 javac HelloWorld。
运行该命令编译程序并将 HelloWorld.class 文件放到 *com/oreilly/tutorial/* 目录中。
4．在包所在的目录中创建名为 Manifest.txt 的文件。在该文件中，包括如下这行内容，指定了 `main` 类位于何处：

```
Main-Class: com.oreilly.tutorial.HelloWorld
```

5．执行 jar cmf Manifest.txt helloWorld.jar com/oreilly/tutorial。
使用这个命令创建 JAR 文件，它会将 Manifest.txt 的内容添加到清单文件 MANIFEST.MF 中。清单文件用来定义扩展和各种包相关的数据：

```
Manifest-Version: 1.0
Created-By: 1.7.0 (Oracle Corporation)
Main-Class: com.oreilly.tutorial.HelloWorld
```

6．运行 jar tf HelloWorld.jar。
使用该命令能够展现 JAR 文件中的内容：

```
META-INF/
META-INF/MANIFEST.MF
com/
com/oreilly/
com/oreilly/tutorial
com/oreilly/tutorial/HelloWorld.class
```

7．最后，运行 java –jar HelloWorld.jar。
使用该命令来执行 JAR 文件。

118

类路径

类路径（主要针对 Java 8）是一个参数集，多个命令行工具都会用到，以便于告知 JVM 去何处查找用户定义的类和包。类路径的约定在不同操作系统下是不同的。

在微软 Windows 下，路径中的目录使用反斜线描述并且使用分号来分割路径：

```
-classpath \home\XClasses\;dir\YClasses\;.
```

在兼容 POSIX 的操作系统中（如 Solaris、Linux 和 macOS），路径中的目录使用斜线描述并且使用冒号来分割路径：

```
-classpath /home/XClasses/:dir/YClasses/:.
```

提示

句点代表了当前工作目录。

我们可以设置 CLASSPATH 环境变量，告诉 Java 编译器去何处查找类文件和包：

```
rem Windows
set CLASSPATH=classpath1;classpath2

# Linux, Solaris, macOS
# (May vary due to shell specifics)
setenv CLASSPATH classpath1:classpath2
```

第 11 章

内存管理

Java 具有自动的内存管理功能，称为垃圾收集（garbage collection，GC）。GC 的主要任务是分配内存、维护内存中对象的引用以及恢复那些不再引用的对象的内存。 119

垃圾收集器

从 J2SE 5.0 发布版本开始，Java HotSpot 虚拟机会自行进行一些优化。这个过程包括在启动的时候基于平台信息尝试选择最合适的 GC 或相关的设置，并且还会进行运行时的调优。

尽管针对 GC 的初始设置和运行时调优总体是非常成功的，但是有时候我们可能想要根据以下目标变更或调整 GC。

最大的暂停时间目标

> 最大的暂停时间目标指的是 GC 停顿应用来恢复内存所需的时间。

吞吐量目标

> 吞吐量目标指的是应用程序所占据的时间，或者说 GC 之外所花费的时间。

下面将介绍了各种垃圾收集器的概要情况、主要关注点以及它 120

们应该在什么场景下使用。在“命令行参数”一节中将介绍手动选择 GC 的信息。

Serial 收集器

Serial 收集器会在单个 CPU 的单个线程上执行。当这个 GC 线程运行的时候，应用的执行将会暂停直至收集过程完成。

这个收集机制最适用于应用具备较小的数据集，最大约为 100MB，并且对低暂停时间没有要求。

Parallel 收集器

Parallel 收集器，又称为吞吐（*Throughput*）收集器，能够跨多个 CPU 在多个线程中执行。使用多个线程能够显著加快 GC 的过程。

这个收集器最适用的场景是没有暂停时间的限制并且应用性能是程序最重要的关注项。

Parallel 整理收集器

Parallel 整理（compacting）收集器类似于 Parallel 收集器，只不过它优化了算法能够减少收集的暂停时间。

这个收集器最适用于具有暂停时间要求的应用。

提示

Parallel 整理收集器是从 J2SE 5.0 update 6 版本开始提供的。

---

121 并发标记清除收集器

并发标记清除收集器（Concurrent Mark-Sweep，CMS）又被称为低延迟收集器（*low-latency collector*），实现了处理大规模集合的算法，这样的场景可能会导致长时间的暂停。

这个收集器适用于响应时间的重要性超过吞吐时间和 GC 暂停的场景。

垃圾优先收集器（G1）

垃圾优先收集器（Garbage-First collector），又被称为 G1 收集器，用于大内存多核心的机器。这种服务器风格的 GC 以较高的概率满足暂停时间的目标，同时还能实现高吞吐。针对整个堆的操作（如全局标记）是与应用线程并行执行的，这样就能避免随着堆或实际数据规模的增加出现同等比例的中断。

提示

在 32 位和 64 位的服务器配置中，Java SE 9 会将 G1 作为默认的垃圾收集器。G1 是从 Java SE 7 update 4 版本开始提供的。

内存管理工具

尽管 GC 本身的优化被证明是非常成功的，但是需要注意的很重要的一点就是 GC 并不能打包票，它只是一个目标。在某个平台上获得的提升换到另外一个平台上可能就没有效果了。最好使用内存管理工具（包括 profiler）找到问题的根源。

表 11-1 列出了这些工具。除了 Heap/CPU Profiling Tool（HPROF）之外，其他的都是命令行应用。HPROF 要通过命令行参数动态加载。

表11-1　JDK内存管理工具

122

资源	描述
jvisualvm	问题排查工具，打包到了 Java 8 中，但是没有包含到 Java 9 中
jconsole	Java 管理扩展（Java Management Extensions，JMX）——兼容监控工具

资源	描述
jinfo	配置信息工具
jstat	JVM 统计监控工具
jstatd	允许远程工具 attach 的 jstat
jmc	分析、监控和诊断工具
jmap	内存映射工具
jstack	栈跟踪工具
jcmd	诊断命令请求的工具
jdb	Java 调试工具
jps	显示 JVM 进程列表的工具

提示

读者可以考虑了解一下 Oracle Java SE 的高级功能，包括 Java Mission Control（也就是 jmc）和 Java Flight Recorder。它们都是企业级、面向生产环境的诊断和监控工具。要在生产环境使用 Java Flight Recorder 的话，需要商业的许可证。

命令行参数

如下 GC 相关的命令行参数传递给 Java 解释器，能够实现与 Java HotSpot 虚拟机功能的交互。

`-XX:+PrintGC` 或 `-verbose:gc`

在每次进行收集的时候，打印出堆和本次收集的基本信息。在 Java 9 中，GC 的日志信息使用了统一的 JVM 日志框架。

`-XX:+PrintCommandLineFlags –version`

123

打印堆设置、所应用的 -XX 的值以及版本信息。

`-XX:+PrintGCDetails`

在每次收集的时候，打印堆和垃圾收集相关的详细信息。

`-XX:+PrintGCTimeStamps`

在 PrintGC 或 PrintGCDetails 的输出上添加时间戳。

`-XX:+UseSerialGC`

启用 Serial 收集器。

`-XX:+UseParallelGC`

启用 Parallel 收集新生代。

`-XX:+UseParallelOldGC`

在新生代和老年代都启用 Parallel 收集器。

`-XX:+UseParNewGC`

启用 ParNew 新生代收集器。可以与并发低延迟收集器协同使用，在 Java 9 中已被移除。

`-XX:+UseConcMarkSweepGC`

启用并发低延迟 CMS 收集器。可以与 Parallel 新生代收集器协同使用，在 Java 9 中已被移除。

`-XX:+UseG1GC`

使用垃圾优先（G1）收集器。

`-XX:+DisableExplicitGC`

禁用显式的 GC（System.gc()）方法。

`-XX:ParallelGCThreads=[`*`threads`*`]`

定义 GC 线程的数量。默认值与 CPU 的数量相关，该参数适用于 CMS 和 Parallel 收集器。

`-XX:MaxGCPauseMillis=[`*`milliseconds`*`]` 124

提供给 GC 的提示信息，设置以毫秒为单位的最大暂停时间目标。这个参数可以用于 Parallel 收集器。

`-XX:+GCTimeRatio=[__value__]`

提供给 GC 的提示信息，设置为了实现吞吐目标，GC 时间与应用时间的比例 (1 / (1 + [value]))。默认值为 99，也就是

应用运行将占据 99% 的时间，因此允许 GC 运行占用 1% 的时间。这个参数能够应用于 Parallel 收集器。

`-XX:+CMSIncrementalMode`

为 CMS 收集器启用增量（incremental）模式。用于一个或两个处理器的机器，在 Java 9 中已被移除。

`-XX:+CMSIncrementalPacing`

为 CMS 收集器启用自动调节功能。

`-XX:MinHeapFreeRatio=[percent]`

设置在 GC 之后，空闲空间与堆的总空间的最小目标百分比。默认百分比是 40。

`-XX:MaxHeapFreeRatio=[percent]`

设置在 GC 之后，空闲空间与堆的总空间的最大目标百分比。默认百分比是 70。

`-Xms[bytes]`

以字节作为单位覆盖默认的最小堆空间。默认值：系统物理内存的 1/64，最大到 1GB。对于非服务器级别的机器，初始的堆大小为 4 MB。

`-Xmx[bytes]`

以字节作为单位覆盖默认的最大堆空间。默认值：小于 1/4 的物理内存或 1 GB。对于非服务器级别的机器，初始的堆大小为 64 MB。

`-Xmn[bytes]`

堆中新生代的大小。

125 `-XX:OnError=[command_line_tool [__options__]]`

指定当出现严重错误时，用户所提供的脚本或命令。

`-XX+AggressiveOpts`

启用在未来释放版本可能会默认开启的性能优化。

关于可用参数的更完整列表，参见 Java HotSpot VM 的参数页面

（*http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html*）。Java 9 会校验命令行标记以避免出现崩溃。

> **提示**
>
> 字节值包括用于 KB 的 [k|K]，用于 MB 的 [m|M] 以及用于 GB 的 [g|G]。

需要注意的是，–XX 参数并不保证其稳定性。它们不是 Java 语言规范（Java Language Specification，JLS）的一部分。如果其他第三方 JVM 也有类似参数，并不能保证它们的形式和功能完全一致。

调整 Java 堆的大小

堆内存区域存储了 Java 程序运行期间所创建的对象。只有当堆所需的区域大于默认值的时候，我们才需要调整它的大小。如果你遇到性能问题或者见到堆（Java heap space）的 java.lang.OutOfMemoryError 错误信息，那么意味着堆空间可能已被耗尽。

元空间

原生内存用来存储类的元数据，这会创建一个名为元空间（*Metaspace*）的内存区域。元空间替代了 PermGen 模型，所以在 JDK 8 HotSpot JVM 之后，将不会再见到 PermGen OutOfMemoryError 发生了。在 Java 9 之前的版本中，提供了 JVisualVM 工具，能够对元空间进行分析，查看是否存在内存泄露。JVisualVM 现在由 VisualVM 在 GitHub（https://visualvm.github.io/）上进行维护。

与 GC 进行交互

126

与垃圾收集器进行交互可以通过显式调用或者覆盖 `finalize` 方

法来实现。

显式垃圾收集

垃圾收集器可以通过 System.gc() 或 Runtime.getRuntime().gc() 方法显式调用。但是，一般来讲，显式调用 GC 是要避免使用的，因为它会强制执行 full 收集（即便是在 minor 就能满足需求的时候），增加不必要的暂停时间。对 System.gc() 的请求并不一定始终会得到满足，因为 JVM 有时会将其忽略。

终结方法

每个对象都有一个从 Object 继承到的 finalize() 方法。垃圾收集器在销毁该对象之前，会调用该方法，但是这个调用的执行时间并不确定。默认的 finalize() 方法什么事情都不会做，尽管不推荐，但是该方法可以覆盖：

```
public class TempClass extends SuperClass {
  ...
  // 当垃圾收集的时候执行
  protected void finalize() throws Throwable {
    try {
      将需要的功能放到这里
    } finally {
      // 我们可以调用超类的 finalize 方法
      super.finalize(); // 来源于父类的方法
    }
  }
}
```

127 如下的样例销毁了该对象：

```
public class MainClass {
  public static void main(String[] args) {
    TempClass t = new TempClass();
    // 将对象的引用移除
    t = null;
    // 可以执行 GC 了
    System.gc();
  }
}
```

第 12 章

基本输入和输出 129

Java 提供了多个基本的输入和输出，我们将会在本章中讨论其中的一部分。基础的类能够用来读取和写入文件、socket 和控制台。它们还提供了操作文件与目录以及序列化数据的功能。Java I/O 所抛出异常（包括 IOException）都需要进行处理。

Java I/O 类还支持格式化数据、压缩和解压缩流、加密和解密以及在线程间使用管道流进行通信的功能。

新 I/O（NIO）是在 Java 1.4 中引入的，它提供了额外的 I/O 功能，包括缓冲、文件锁定、正则表达式匹配、可扩展的网络以及缓冲管理。

NIO.2 是在 Java SE 7 中引入的，该话题将会在下一章进行讨论。NIO.2 扩展了 NIO 并提供了新的文件系统。

标准的流 in、out 和 err

Java 使用了 3 个标准流：in、out 和 err。

System.in 是标准的输入流，它用来从用户获取数据到程序之中： 130

```
byte teamName[] = new byte[200];
int size = System.in.read(teamName);
System.out.write(teamName,0,size);
```

System.out 是标准的输出流，用来从程序输出数据给用户：

```
System.out.print( “Team complete” );
```

System.err 是标准的错误流，用来将错误数据从程序输出给用户：

```
System.err.println( “Not enough players” );
```

需要注意的是，这些方法都可能会抛出 IOException。

提示

Java SE 6 所引入的 Console 类提供了一个标准流的备选方案，实现与命令行环境的交互。

标准输入和输出类的层级结构

图 12-1 展现了常用的 Reader、Writer 以及输入输出流的层级结构。注意，I/O 可以链接起来实现多重效果。

131

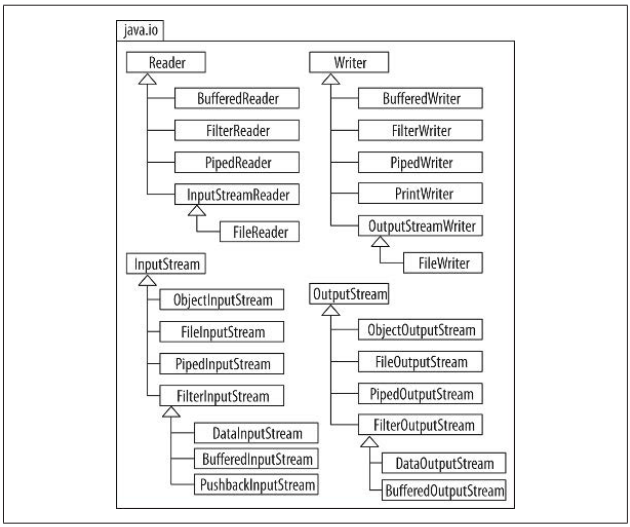


图12-1 常用的Reader、Writer以及输入输出流

Reader 和 Writer 类用来读取和写入字符数据（文本）。InputStream 和 OutputStream 一般用来写入二进制数据。

文件读取和写入

Java 提供了多种基础实施简化系统文件的读取和写入。

从文件中读取字符数据

从文件中读取字符数据时，可以使用 BufferedReader。另外还可以使用 FileReader，但是如果数据量比较大，效率不会那么高。调用 readLine() 方法将会从文件中读取一行文本。当读取完成时，要调用 BufferedReader 的 close() 方法：

```
BufferedReader bReader = new BufferedReader
     (new FileReader( "Master.txt" ));
String lineContents;
while ((lineContents = bReader.readLine())
      != null) {...}
bReader.close();
```

132

可以使用 NIO 2.0 的 Files.newBufferedReader(<path>,<charset>); 方法指定文件编码，避免隐式地对文件编码假定。

从文件中读取二进制数据

要读取二进制数据，可以使用 DataInputStream。调用 read() 会从输入流中读取数据。如果是要读取一个字节数组，可以直接使用 InputStream：

```
DataInputStream inStream = new DataInputStream
      (new FileInputStream( "Team.bin" ));
inStream.read();
```

如果要读取大量的数据，那么应该使用 BufferedInputStream，从而让数据的读取更加高效：

```
DataInputStream inStream = new DataInputStream
(new BufferedInputStream(new FileInputStream(team)));
```

通过使用 PushbackInputStream 类中的方法，已经读取的二进

制数据可以重新放回到流中：

```
unread(int i);    // 放回一个字节
unread(byte[] b); // 放回一个字节数组
```

写入字符数据到文件中

133 要写入字符到文件中，可以使用 PrintWriter。当写入输出流完成的时候，需要调用 PrintWriter 类的 close() 方法：

```
String in = "A huge line of text";
PrintWriter pWriter = new PrintWriter
  (new FileWriter( "CoachList.txt" ));
pWriter.println(in);
pWriter.close();
```

如果只有少量文本要写入文件，也可以使用 FileWriter。需要注意，如果传递给 FileWriter 的文件不存在，将会自动创建：

```
FileWriter fWriter = new
  FileWriter( "CoachList.txt" );
fWriter.write( "This is the coach list." );
fWriter.close();
```

写入二进制数据到文件中

要写入二进制数据，可以使用 DataOutputStream。调用 writeInt() 将会写入一个整型数组到输出流中：

```
File positions = new File( "Positions.bin" );
int[] pos = {0, 1, 2, 3, 4};
DataOutputStream outStream = new DataOutputStream
   (new FileOutputStream(positions));
for (int i = 0; i < pos.length; i++)
  outStream.writeInt(pos[i]);
```

要写入大量数据，可以使用 BufferedOutputStream：

```
DataOutputStream outStream = new DataOutputStream
(new BufferedOutputStream(positions));
```

Socket 读取和写入

Java 提供了一些基础设施，简化系统 Socket 的读取和写入。

从 Socket 中读取字符数据

要从 Socket 中读取字符数据，可以连接至该 Socket 然后使用 BufferedReader 来读取数据：

```
Socket socket = new Socket( “127.0.0.1” , 64783);
InputStreamReader reader = new InputStreamReader
    (socket.getInputStream());
BufferedReader bReader
  = new BufferedReader(reader);
String msg = bReader.readLine();
```

134

BufferedReader 在 Java SE 8 中引入了 lines() 方法，以适应新的 Stream API。该方法返回一个 Stream，其中的元素是行数据，会以懒加载的方式从上下文 BufferedReader 中进行读取。

从 Socket 中读取二进制数据

要读取二进制数据，需要使用 DataInputStream，调用 read() 方法会从输入流中读取数据。注意 Socket 类位于 java.net 包中：

```
Socket socket = new Socket( “127.0.0.1” , 64783);
DataInputStream inStream = new DataInputStream
   (socket.getInputStream());
inStream.read();
```

如果需要读取大量数据，那么请使用 BufferedInputStream，它能够让数据的读取更加高效：

```
DataInputStream inStream = new DataInputStream
(new BufferedInputStream(socket.getInputStream()));
```

写入字符数据到 Socket 中

要写入字符数据到 Socket 中，可以连接至该 Socket 然后创建并使用 PrintWriter 来写入字符数据：

```
Socket socket = new Socket( “127.0.0.1” , 64783);
PrintWriter pWriter = new PrintWriter
   (socket.getOutputStream());
pWriter.println( “Dad, we won the game.” );
```

写入二进制数据到 Socket 中

要写入二进制数据，可以使用 DataOutputStream，调用其 write() 方法将数据写入到输出流中：

```
byte positions[] = new byte[10];
Socket socket = new Socket( “127.0.0.1” , 64783);
DataOutputStream outStream = new DataOutputStream
  (socket.getOutputStream());
outStream.write(positions, 0, 10);
```

135

如果要写入大量数据的话，那么同样可以使用 BufferedOutputStream：

```
DataOutputStream outStream = new DataOutputStream
(new BufferedOutputStream(socket.getOutputStream()));
```

序列化

要将某个对象（以及它需要还原的所有关联数据）保存为字节数组，这个对象的类必须要实现 Serializable 接口。需要注意的是，声明为 transient 的数据成员不会包含在序列化的对象中。使用序列化和反序列化要非常小心，因为类的变更，包括在类层级结构上进行转移、删除字段、将某个字段声明为非 transient 或静态以及使用不同的 JVM，都将会影响到对象的恢复。

ObjectOutputStream 和 ObjectInputStream 类可以用来序列化和反序列化对象。

序列化

要序列化一个对象，需要使用 ObjectOutputStream：

```
ObjectOutputStream s = new
  ObjectOutputStream(new FileOutputStream( “p.ser” ));
```

使用序列化功能的样例如下所示：

```
ObjectOutputStream oStream = new
  ObjectOutputStream(new
  FileOutputStream( “PlayerDat.ser” ));
```

```
oStream.writeObject(player);
oStream.close();
```

反序列化

136

要反序列一个对象（也就是从对象的扁平化版本转换为正常的对象），需要使用 ObjectInputStream，读入文件并将数据转型为对应的对象：

```
ObjectInputStream d = new
  ObjectInputStream(new FileInputStream( “p.ser” ));
```

反序列化的一个例子如下所示：

```
ObjectInputStream iStream = new
    ObjectInputStream(new
    FileInputStream( “PlayerDat.ser” ));
Player p = (Player) iStream.readObject();
```

Java 9 允许对对象序列化数据所传入的流进行过滤，从而实现安全性和健壮性。

压缩和解压文件

Java 提供了创建压缩 ZIP 和 GZIP 文件的类。ZIP 会归档多个文件，而 GZIP 则是归档一个文件。

我们可以使用 ZipOutputStream 来压缩文件，使用 ZipInputSteam 来解压文件：

```
ZipOutputStream zipOut = new ZipOutputStream(
    new FileOutputStream( “out.zip” ));
String[] fNames = new String[] { “f1” , “f2” };
for (int i = 0; i < fNames.length; i++) {
ZipEntry entry = new ZipEntry(fNames[i]);
FileInputStream fin =
    new FileInputStream(fNames[i]);
try {
  zipOut.putNextEntry(entry);
  for (int a = fin.read();
    a != -1; a = fin.read()) {
        zipOut.write(a);
  }
  fin.close();
```

```
        zipOut.close();
137   } catch (IOException ioe) {...}
    }
```

要解压文件，需要创建 ZipInputStream 并调用其 getNextEntry() 方法，将文件读取到一个 OutputStream 中。

压缩和解压缩 GZIP 文件

要压缩 GZIP 文件，需要创建一个新的 GZIPOutputStream，将带有“.gzip”扩展的文件名传给它，然后将来自 GZIPOutputStream 的数据传输给 FileInputStream。

要解压 GZIP 文件，创建一个 GZipInputStream 并创建一个 FileOutputStream，将数据读取到输出流中。

第 13 章

新I/O API（NIO.2）

NIO.2 是在 JDK 7 引入的，它提供了增强的文件 I/O 支持并允许访问默认的文件系统。NIO.2 是由 java.nio.file 和 java.nio.file.attribute 包来提供支持的。NIO.2 API 也被称为 *JSR 203: More New I/O APIs for the Java Platform*。在该 API 中常用的接口是 Path、PathMatcher、FileVisitor 和 WatchService，常用的类是 Paths 和 Files。

Path 接口

Path 接口可以用来操作文件和目录路径。这个类是 java.io.File 类的升级版本。如下的代码阐述了 Path 接口和 Paths 类的一些方法的具体用法，我们在这里使用它们来获取信息：

```
Path p = Paths.get("\\opt\\jpgTools\\README.txt");
System.out.println(p.getParent()); // \opt\jpgTools
System.out.println(p.getRoot()); // \
System.out.println(p.getNameCount()); // 3
System.out.println(p.getName(0)); // opt
System.out.println(p.getName(1)); // jpgTools
System.out.println(p.getFileName()); // README.txt
System.out.println(p.toString()); // The full path
```

Path 类还提供了额外的特性，其中一部分如表 13-1 所示。

表13-1　Path接口的功能

Path 方法	功能
path.toUri()	将一个 Path 转换为 URI 对象
path.resolve(Path)	将两个 Path 连接在一起
path.relativize(Path)	根据当前的位置和另外一个位置，构建一个相对 Path
path.compareTo(Path)	比较两个 Path

Files 类

Files 类能够用来创建、检查、删除、复制或移动某个文件或目录。如下的样例展现了 Files 类一些常用的方法：

```
// 创建目录
Path dirs = Paths.get( "\\opt\\jpg\\" );
Files.createDirectories(dirs);
// 初始化 Path 对象
Path target1 = Paths.get( "\\opt\\jpg\\README1.txt" );
Path p1 = Files.createFile(target1);
Path target2 = Paths.get( "\\opt\\jpg\\README2.txt" );
Path p2 = Files.createFile(target2);
// 检查文件属性
System.out.println(Files.isReadable(p1));
System.out.println(Files.isReadable(p2));
System.out.println(Files.isExecutable(p1));
System.out.println(Files.isSymbolicLink(p1));
System.out.println(Files.isWritable(p1));
System.out.println(Files.isHidden(p1));
System.out.println(Files.isSameFile(p1, p2));

// 删除、移动和复制的样例
Files.delete(p2);
System.out.println(Files.move(p1, p2));
System.out.println(Files.copy(p2, p1));
Files.delete(p1);
Files.delete(p2);
```

141

方法 move 能够接受可变长度参数的枚举值 REPLACE_EXISTING 或 ATOMIC_MOVE。如果使用 REPLACE_EXISTING，即便文件已经存在，也会进行移动。ATOMIC_MOVE 能够确保任何监控（watch）该目录的进程都能访问到完整的文件。

方法 copy 能够接受可变长度参数的枚举值 REPLACE_EXISTING、COPY_ATTRIBUTES 或 NOFOLLOW_LINKS。其中，如果使用 REPLACE_EXISTING，即便文件已经存在，也会进行复制。COPY_ATTRIBUTES 会复制文件的属性。NOFOLLOW_LINKS 会复制链接，而不是目标本身。

Files 类添加了 lines、list、walk 和 find 方法以适应 Stream API。lines 方法会以懒加载的方式读取行数据的流。list 方法以懒加载的方式列出目录条目，而 walk 方法会递归遍历所有的条目。find 方法则会基于一个给定的文件节点，在文件树中以懒加载的方式搜索文件并以 Path 接口的方式提供结果。

其他特性

NIO 2.0 API 还提供了如下的特性，了解它们对于我们的工作是有好处的。关于这些特性的问题，也包含在 Oracle Certified Professional Java SE 7 Programmer 考试中。这些内容在这里不会详细介绍，因为它们更适合教程方式的指南或资源：

- 使用 WatchService 能够实现目录的监控（watch）；
- 使用 FileVisitor 接口递归访问目录树；
- 使用 PathMatcher 函数式接口查找文件。

因为 PathMatcher 是一个函数式接口，所以它可以按照 lambda 142 表达式的方式来使用：

```
PathMatcher matcher = (Path p) -> {
  // returns boolean
  return (p.toString().contains( “World” ));
};
Path path1 =  FileSystems.getDefault().getPath(
  “\\tmp\\Hello.java” );
Path path2 =  FileSystems.getDefault().getPath(
  “\\tmp\\HelloWorld.java” );
System.out.print( “Matches: “
  + matcher.matches(path1) + “, “
  + matcher.matches(path2));

$ Matches: false, true
```

提示

在遍历某个目录时，可以考虑组合使用新的java.nio.file.Directory Stream函数式接口和增强的for循环。

第 14 章

并发 143

在 Java 中，线程允许使用多个处理器或者一个处理器中的多个核心，从而实现更加高效的计算处理。在单个处理器上，线程提供了并发操作比如重叠的 I/O（overlapping I/O）处理。

Java 以 Thread 类和 Runnable 接口的方式提供多线程的支持。

创建线程

我们有两种方式来创建线程，要么扩展 `java.lang.Thread` 类，要么实现 `java.lang.Runnable` 接口。

扩展 Thread 类

扩展 Thread 类并覆盖 `run()` 方法能够创建一个线程化的类。如下以一种简便的方式启动了一个线程：

```
class Comet extends Thread {
  public void orbit() {
    System.out.println( “orbiting” );
  }
  public void run() {
    orbit();
  }
}
```

144

```
Comet halley = new Comet();
halley.start();
```

需要注意的是，在 Java 中某个类只能扩展一个超类，所以扩展了 Thread 的类就不能扩展其他超类了。

实现 Runnable 接口

实现 Runnable 接口并定义其 run() 方法也可以创建线程化的类：

```
class Asteroid implements Runnable {
  public void orbit() {
    System.out.println( “orbiting” );
  }
  public void run() {
    orbit();
  }
}

Asteroid majaAsteroid = new Asteroid();
Thread majaThread = new Thread(majaAsteroid);
majaThread.start();
```

可运行的实例能够传递给多个线程对象。每个线程会执行相同的任务，下面展示了使用 lambda 表达式后的代码：

```
Runnable asteroid = () -> {
  System.out.println( “orbiting” );
};
Thread asteroidThread1 = new Thread(asteroid);
Thread asteroidThread2 = new Thread(asteroid);
asteroidThread1.start();
asteroidThread2.start();
```

线程状态

145 枚举 Thread.state 提供了 6 种线程状态，如表 14-1 所示。

表14-1 线程状态

线程状态	描述
NEW	线程已经创建但是还没有启动
RUNNABLE	线程可以运行
BLOCKED	线程“存活（alive）”，但处于阻塞，在等待一个监视器锁

续表

线程状态	描述
WAITING	线程“存活（alive）”，调用了自己的 wait() 或 join() 方法，在等待其他的线程
TIMED_WAITING	线程“存活（alive）”，在特定的时间段内等待另外一个线程；休眠
TERMINATED	线程已经完成

线程优先级

优先级的合法值一般在 1 到 10 之间，默认值是 5。线程优先级是 Java 中最不具有迁移性的特性之一，因为它们的范围和默认值在不同 Java 虚拟机（Java Virtual Machine，JVM）上是有差异的。使用 MIN_PRIOR ITY、NORM_PRIORITY 和 MAX_PRIORITY 可以获取优先级的值：

```
System.out.print(Thread.MAX_PRIORITY);
```

低优先级的线程执行时要让位于高优先级的线程。

常用方法

表 14-2 列出了 Thread 类中供线程使用的常见方法。

表14-2 线程方法

146

方法	描述
getPriority()	返回线程的优先级
getState()	返回线程的状态
interrupt()	中断线程
isAlive()	返回线程的存活状态
isInterrupted()	检查线程是否中断
join()	调用该方法的线程会等待本对象所代表的线程完成为止
setPriority(int)	设置线程的优先级
start()	将线程置于运行状态

表 14-3 列出了 Object 类中与线程相关的常用方法。

表14-3　Object类中用于线程的方法

方法	描述
`notify()`	通知一个线程恢复并运行
`notifyAll()`	通知所有等待某个线程或资源的线程恢复，然后调度器将会选择其中的一个线程运行
`wait()`	暂停某个线程到等待状态，直到另外一个线程调用 `notify()` 或 `notifyAll()`

提示

如果线程已经将中断标记置为 `true`，调用 `wait()` 和 `notify()` 会抛出 `InterruptedException`。

147 表 14-4 中列出了 Thread 类中实现线程相关功能的静态方法（如 `Thread.sleep(1000)`）。

表14-4　静态线程方法

方法	描述
`activeCount()`	返回当前线程所在的组中线程的数量
`currentThread()`	返回当前正在运行的线程的引用
`interrupted()`	检查当前正在运行的线程的中断状态
`sleep(long)`	将当前正在运行的线程阻塞参数所指定的毫秒数
`yield()`	暂停当前的线程，允许其他线程运行

同步

`synchronized` 关键字提供了一种方式将锁应用到代码块和方法上。锁应当用到访问关键共享资源的代码块和方法上。这些监视器锁以大括号作为开始和结束。如下的代码展现了同步代码

块和方法的样例。

对象实例 t 使用同步锁：

```
synchronized (t) {
  // Block body
}
```

对象实例 this 使用同步锁：

```
synchronized (this) {
  // Block body
}
```

方法 raise() 使用同步锁：

```
// 等价的代码片段 1
synchronized void raise() {
  // Method body
}

// 等价的代码片段 2
void raise() {
  synchronized (this) {
    // Method body
  }
}
```

148

静态方法 calibrate() 使用同步锁：

```
class Telescope {
  synchronized static void calibrate() {
    // Method body
  }
}
```

提示

锁也被称为监视器（monitor）或互斥（mutex，手动排他锁）。

并发工具集提供了额外的方式来应用和管理并发。

并发工具集

Java 2 SE 5.0 为并发编程提供了工具类。这些工具类位于 java.util.concurrent 包中，它们包括执行器（executor）、并发集合、同步器以及时间工具类。

执行器

ThreadPoolExecutor 类及其子类 ScheduledThreadPoolExecutor 实现了 Executor 接口，能够提供可配置、灵活的线程池。线程池允许服务器组件利用线程的可重用性。

149 Executors 类提供了工厂（对象创建器）方法和工具方法。其中，如下的方法是用来创建线程池的：

`newCachedThreadPool()`

创建一个无界的线程池，它会自动重用线程。

`newFixedThreadPool(int nThreads)`

创建一个固定大小的线程池，它会基于一个共享的无界队列重用线程。

`newScheduledThreadPool(int corePoolSize)`

创建一个线程池，它能够让命令阶段性或基于一定的延迟调度执行。

`newSingleThreadExecutor()`

创建一个单线程的执行器，它会基于一个无界的队列进行操作。

`newSingleThreadScheduledExecutor()`

创建一个单线程的执行器，它能够让命令阶段性或基于一定的延迟调度执行。

如下的样例展现了如何使用 newFixedThreadPool：

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
```

```
public class ThreadPoolExample {
  public static void main() {
    // 创建任务
    // （代码来源于 “class RTask implements Runnable”）
    RTask t1 = new RTask( “thread1” );
    RTask t2 = new RTask( “thread2” );

    // 创建线程管理器
    ExecutorService threadExecutor =
        Executors.newFixedThreadPool(2);

    // 让线程可运行
    threadExecutor.execute(t1);
    threadExecutor.execute(t2);

    // 关闭线程
    threadExecutor.shutdown();
  }
}
```

150

并发集合

虽然集合类型能够进行同步，但是我们最好还是使用并发线程安全的类来执行相同的功能，如表 14-5 所示。

表14-5　集合及其线程安全的对等类

集合类	线程安全的对等类
HashMap	ConcurrentHashMap
TreeMap	ConcurrentSkipListMap
TreeSet	ConcurrentSkipListSet
Map 子类型	ConcurrentMap
List 子类型	CopyOnWriteArrayList
Set 子类型	CopyOnWriteArraySet
PriorityQueue	PriorityBlockingQueue
Deque	BlockingDeque
Queue	BlockingQueue

同步器

同步器（synchronizer）是满足特定目的的同步工具。可用的同

步器如表 14-6 所示。

表14-6　同步器

同步器	描述
Semaphore	维护一组许可
CountDownLatch	基于一组正在执行的操作实现等待功能
CyclicBarrier	基于一个公用的栅栏点实现等待功能
Exchanger	实现一个同步点，线程可以在这个点交换元素

时间工具类

TimeUnit 枚举通常用来通知基于时间的方法该如何计算时间参数，如下面的样例所示：

```
// tyrLock (long time, TimeUnit unit)
if (lock.tryLock(15L, TimeUnit.DAYS)) {...}
  //15 days
```

可用的 TimeUnit 枚举常量如表 14-7 所示。

表14-7　TimeUnit常量

常量	单位定义	单位（以秒为单位计算）	缩写
NANOSECONDS	1/1000μs	.000000001	ns
MICROSECONDS	1/1000 ms	.000001	μs
MILLISECONDS	1/1000 sec	.001	ms
SECONDS	sec	1	sec
MINUTES	60 sec	60	min
HOURS	60 min	3600	hr
DAYS	24 hr	86400	d

第 15 章

Java集合框架

Java 集合框架的设计目标是以层级结构的形式支持各种各样的集合。它实际是由接口、实现以及算法所组成的 。

Collection 接口

集合是对多个元素进行分组的对象，它们还能对这些元素进行存储、检索和操作。Collection 接口位于集合层级结构的根部。Collection 的子接口包括 List、Queue 和 Set。表 15-1 展示了这些接口以及它们是否有序和允许重复。Map 接口也包含在了这里，因为它是该框架的一部分。

表15-1　常见的集合

接口	是否有序	是否允许重复	备注
List	是	是	基于位置的访问、元素插入控制
Map	可以实现	否（key 不允许重复）	Key 是唯一的；每个 key 最多映射一个值
Queue	是	是	持有元素；通常使用 FIFO 算法
Set	可以实现	否	保证唯一性

154 实现

表 15-2 列出了常用的集合类型实现以及它们的接口、是否有序、是否能够排序以及是否允许包含重复的值。

表15-2　集合类型的实现

实现	接口	是否有序	是否支持排序	是否允许重复	备注
ArrayList	List	索引	否	是	快速重新调整大小的数组
LinkedList	List	索引	否	是	双向链表
Vector	List	索引	否	是	遗留的结构、同步的
HashMap	Map	否	否	否	键 / 值对
Hashtable	Map	否	否	否	遗留的数据结构、同步的
LinkedHashMap	Map	插入顺序、最后访问	否	否	链表 /hash 表
TreeMap	Map	平衡	是	否	红黑树 Map
PriorityQueue	Queue	优先级	是	是	堆实现
HashSet	Set	否	否	否	快速访问的集合
LinkedHashSet	Set	插入顺序	否	否	链表 /hash 集
TreeSet	Set	有序	是	否	红黑树 Set

155 集合框架方法

Collection 接口的子接口提供了一些有价值的方法签名，如表 5-3 所示。

表15-3　有价值的子接口方法

方法	List 的参数	Set 的参数	Map 的参数	返回值
add	index, element	element	n/a	boolean
contains	Object	Object	n/a	boolean
containsKey	n/a	n/a	key	boolean
containsValue	n/a	n/a	value	boolean
get	index	n/a	key	Object
indexOf	Object	n/a	n/a	int
iterator	none	none	n/a	Iterator
keySet	n/a	n/a	none	Set
put	n/a	n/a	key,value	void
remove	index 或 Object	Object	key	void
size	none	none	none	int

Collection.stream() 会返回一个串行（sequential）的流，并将上下文集合作为它的源；Collection.parallelStream() 会返回一个并行（parallel）流，并将上下文集合作为它的源。

集合类的算法

Collections 类，请不要将它与 Collection 接口混淆，包含了多个有价值的静态方法（如算法）。这些方法可以针对各种集合类型进行调用。表 15-4 展现了 Collections 类的常见方法以及它们的接受参数和返回值。

表15-4　Collections类的算法

156

方法	参数	返回值
addAll	Collection <? super T>, T...	boolean
max	Collection, [Comparator]	<T>

方法	参数	返回值
min	Collection, [Comparator]	<T>
disjoint	Collection, Collection	boolean
frequency	Collection, Object	int
asLifoQueue	Deque	Queue<T>
reverse	List	void
shuffle	List	void
copy	List destination, List source	void
rotate	List, int distance	void
swap	List, int position, int position	void
binarySearch	List, Object	int
fill	List, Object	void
sort	List, Object, [Comparator]	void
replaceAll	List, Object oldValue, Object newValue	boolean
newSetFromMap	Map	Set<E>

参见第 16 章了解类型化参数的更多信息（如 <T>）。

算法的效率

算法和数据结构会针对不同的原因进行优化：有些是为了随机的元素访问或插入 / 删除，有些是为了保证有序。根据你的需求不同，可能需要切换算法和数据结构。

常见的集合算法、所对应的类型以及平均时间效率如表 15-5 所示。

157 表15-5 算法效率

算法	具体类型	时间
get、set	ArrayList	0 (1)
add、remove	ArrayList	0 (n)

算法	具体类型	时间
contains、indexOf	ArrayList	O (n)
get、put、remove、containsKey	HashMap	O (1)
add、remove、contains	HashSet	O (1)
add、remove、contains	LinkedHashSet	O (1)
get、set、add、remove（从任意一端）	LinkedList	O (1)
get、set、add、remove（根据索引）	LinkedList	O (n)
contains、indexOf	LinkedList	O (n)
peek	PriorityQueue	O (1)
add、remove	PriorityQueue	O (log n)
remove、get、put、containsKey	TreeMap	O (log n)
add、remove、contains	TreeSet	O (log n)

大写字母 O 用来代表时间效率，而 n 代表了元素的数量（参见表 15-6）。

表15-6　大写字母O的含义

符号	描述
O (1)	时间是固定的，跟元素数量无关
O (n)	时间随着元素的数量线性变化
O (log n)	时间与元素数量的对数相关
O (nlog n)	时间与元素数量的线性对数相关

Comparator 函数式接口

158

在 Collections 类中，有些方法会假设集合中的对象是可对比的。如果没有原生顺序，那么可以使用一个实现了 Comparator 函数式接口的辅助类来指定对象如何排序。下面的样例展现了如何对姓氏所生成的变音代码（metaphone code）进行排序：

提示

请参考 Java 编写的变音代码计算器（*http://gliesians.com/metaphone-calculator.faces*）以更好地理解变音代码。

```
import org.apache.commons.codec.language.Metaphone;
public class MetaphoneCode {
  private String metaphoneCode;
  public MetaphoneCode(String surname) {
    Metaphone m = new Metaphone();
    metaphoneCode = m.metaphone(surname) + "( " +
surname + ")" ;
  }
  public String getMetaphoneCode() {
    return metaphoneCode;
  }
  public void setMetaphoneCode(String metaphoneCode) {
    this.metaphoneCode = metaphoneCode;
  }
  public String toString() {
    return this.metaphoneCode;
  }
}

import java.util.Comparator;
public class SurnameSort implements Comparator
<MetaphoneCode> {
  @Override
159  public int compare (MetaphoneCode mc1, MetaphoneCode
mc2) {
    return mc1.getMetaphoneCode().compareTo(mc2.
getMetaphoneCode());
  }
}

import java.util.ArrayList;
import java.util.Collections;
public class SurnameApp {
  public static void main(String[] args) {
    MetaphoneCode m1 = new MetaphoneCode( "Whitede" );
    MetaphoneCode m2 = new
MetaphoneCode( "Whitehead" );
    MetaphoneCode m3 = new MetaphoneCode( "Whitted" );
    MetaphoneCode m4 = new
MetaphoneCode( "Whitshead" );
```

```
        MetaphoneCode m5 = new MetaphoneCode( “White” );
        ArrayList<MetaphoneCode> mlist = new ArrayList
    <>();
        mList.add(m1);
        mList.add(m2);
        mList.add(m3);
        mList.add(m4);
        mList.add(m5);
        System.out.println( “Unsorted: “ + mList );
        SurnameSort cSort = new SurnameSort();
        Collections.sort(mList, cSort);
        System.out.println( “Sorted: “ + mList );
      }
    }

    $ Unsorted: [WTT (Whitede), WTHT( Whitehead), WTT
    (Whitted), WTXT (Whitshead), WT (White)]
    $ Sorted: [WT (White), WTHT (Whitehead), WTT (Whitede),
    WTT (Whitted), WTXT (Whitshead)]
```

SurnameSort类实现了Comparator接口，供cSort实例使用。除此之外，我们还可以使用内部类，避免创建单独的SurnameSort类： 160

```
// SurnameSort cSort = new SurnameSort();
Comparator<MetaphoneCode> cSort = new
Comparator<MetaphoneCode>() {
  public int compare(MetaphoneCode mc1, MetaphoneCode
mc2) {
    return mc1.getMetaphoneCode().compareTo(mc2.
getMetaphoneCode());
  }
};
```

因为Comparator是一个函数式接口，我们还可以使用lambda表达式让代码更加易读：

```
Comparator<MetaphoneCode> cSort = (MetaphoneCode mc1,
MetaphoneCode mc2)
  -> mc1.getMetaphoneCode().compareTo(mc2.
getMetaphoneCode());
```

在参数列表中，类名并不需要显式声明，因为lambda表达式能够知道目标类型的信息。所以，请注意(mc1, mc2)与(MetaphoneCode mc1, MetaphoneCode mc2)：

```
// Example 1
Comparator <MetaphoneCode> cSort = (mc1, mc2)
  -> mc1.getMetaphoneCode().compareTo(mc2.
getMetaphoneCode());
Collections.sort(mList, cSort);

// Example 2
Collections.sort(mList, (mc1, mc2)
 -> mc1.getMetaphoneCode().compareTo(mc2.
getMetaphoneCode()));
```

161 便利的工厂方法

JDK 9 引入了新的便利的工厂方法，它们能够创建不可变的集合（如 List、Set、Map）实例。因此，多行代码可以重构为一行：

```
// 在 Java 9 之前不可变列表的初始化
List<String> haplogroups = new ArrayList<>();
haplogroups.add( “I2” );
haplogroups.add( “I2B” );
haplogroups.add( “IJ” );
haplogroups = Collections.
unmodifiableList(haplogroups);

// 重构为 Java 9 的不可变列表的初始化
List <String> haplogroups = List.of(“I2”,”I2B”, “IJ”);
```

第 16 章

泛型框架 163

泛型框架，在 Java SE 5.0 中引入并在 Java SE 7 和 8 中进行了更新，提供了类型参数化的功能。原始类型的泛型预计会在 Java 10 中提供。

泛型所带来的好处就是明显减少编写库时的代码量。另外一个好处是在很多场景下减少类型转换。

集合框架中的类、Class 类以及其他的 Java 库都已经包含了泛型的功能。

关于泛型框架的综合介绍，请参见 Philip Wadler 和 Maurice Naftalin 合著的 *Java Generics and Collections*（O'Reilly，2009，*http://shop.oreilly.com/product/9780596527754.do*）。

泛型类与接口

泛型类和接口通过在尖括号（即 <T>）中添加类型参数实现参数化类型的功能。这个类型会基于尖括号中的参数进行实例化。

在实例化之后，泛型参数类型将会应用到这个类中具有相同类型的所有方法中。在下面的例子中，`add()` 和 `get()` 方法分别 164
使用参数类型作为它们的传入参数和返回类型：

```
public interface List <E> extends Collection<E>{
  public boolean add(E e);
  E get(int index);
}
```

当声明一个参数化类型的变量时，在类型参数的地方（如 <E>）需要指定具体类型（如 <Integer>）。

随后，当我们从集合中获取元素时，就可以省略类型转换了：

```
// 带有泛型的 List/ArrayList 集合
List<Integer> iList = new ArrayList<Integer>();
iList.add(1000);
// 没有必要进行显式的类型转换
Integer i = iList.get(0);

// 不带泛型的 List/ArrayList 集合
List iList = new ArrayList();
iList.add(1000);
// 需要进行显式的类型转换
Integer i = (Integer)iList.get(0);
```

Java SE 7 引入了钻石操作符（diamond operator）<>，能够减少额外的输入，从而简化泛型类型的创建：

```
// 不使用钻石操作符
List<Integer> iList1 = new ArrayList<Integer>();
// 使用钻石操作符
List<Integer> iList2 = new ArrayList<>();
```

165 具有泛型的构造器

泛型类的构造器不需要泛型类型参数作为入参：

```
// 泛型类
public class SpecialList <E> {
  // 无参的构造器
  public SpecialList() {...}
  public SpecialList(String s) {...}
}
```

这个类的泛型对象可以按照如下的方式创建：

```
SpecialList<String> b = new
        SpecialList<String>();
```

如果泛型类的构造器包含参数类型，比如 String，那么泛型对象可以按照如下的方式进行实例化：

```
SpecialList<String> b = new
     SpecialList<String>( “Joan Marie” );
```

替换原则

按照 *Java Generics and Collections*（O’Reilly）的定义，替换原则（substitution principle）允许在超类参数化的地方使用子类型：

- 给定类型的变量可以设置为该类型任意子类型的值；
- 以给定类型为参数的方法可以基于该类型的子类型作为入参进行调用。

Byte、Short、Integer、Long、Float、Double、BigInteger 和 BigDecimal 都是 Number 类的子类型：

```
// 以通用的 Number 类型声明 List
List<Number> nList = new ArrayList<Number>();
nList.add((byte)27);     // Byte（自动装箱）
nList.add((short)30000); // Short                                    166
nList.add(1234567890);   // Integer
nList.add((long)2e62);   // Long
nList.add((float)3.4);   // Float
nList.add(4000.8);       // Double
nList.add(new BigInteger( “9223372036854775810” ));
nList.add(new BigDecimal( “2.1e309” ));

// Print Number’s subtype values from the list
for( Number n : nList  )
  System.out.println(n);
```

类型参数、通配符与边界

声明泛型类的最简单方式就是使用无界的类型参数，比如 T：

```
public class GenericClass <T> {...}
```

边界（限制）和通配符可以应用到类型参数上，如表 16-1 所示。

表16-1　类型参数、边界和通配符

类型参数	描述
<T>	无界的类型，等价于<T extends Object>
<T,P>	无界的类型，等价于<T extends Object>和<P extends Object>
<T extends P>	上界（upper bounded）类型，特定的类型T是P的子类型
<T extends P & S>	上界类型，特定的类型T是P的子类型，并且实现了类型S
<T super P>	下界（lower bounded）类型，特定的类型T是P的超类型
<?>	无界通配符，任意的对象类型，等价于<? extends Object>
<? extends P>	有界通配符，用来代指类型P的一些未知的子类型
<? extends P & S>	有界通配符，一些未知的类型，需要是P的子类型并且要实现S类型
<? super P>	下界通配符，用来代指类型P的一些超类型

167

Get 和 Put 原则

同样是出自 *Java Generics and Collections* 一书，Get 和 Put 原则详细阐述了 extends 和 super 通配符的最佳用法：

- 如果只是要从一个结构体中获取值，那就使用 extends 通配符；
- 如果只是要往一个结构体中设置值，那就使用 super 通配符；
- 如果要同时为结构体获取和设置值，那就不要使用通配符。

在 List 集合的 addAll() 方法声明中，用到了 extends 通配符，因为这个方法会从集合中 Get 值：

```
public interface List <E> extends Collection<E>{
  boolean addALL(Collection <? extends E> c)
}

List<Integer> srcList = new ArrayList<Integer>();
srcList.add(0);
srcList.add(1);
srcList.add(2);
// 借助 extends 通配符使用 addAll() 方法
List<Integer> destList = new ArrayList<Integer>();
destList.addAll(srcList);
```

在 Collections 类的 addAll() 方法声明中，用到了 super 通配符，因为这个方法会将值 Put 到集合中：

```
public class Collections {
  public static <T> boolean addAll                    <168>
      (Collection<? super T> c, T... elements){...}
}

// 借助 super 通配符使用 addAll() 方法
List<Number> sList = new ArrayList<Number>();
sList.add(0);
Collections.addAll(sList, (byte)1, (short)2);
```

泛型具体化

泛型类型能够以多种方式进行扩展。

以泛型化的抽象类 AbstractSet <E> 为例：

`class SpecialSet<E> extends AbstractSet<E> {...}`

SpecialSet 类使用参数类型 E 扩展了 AbstractSet。这是声明泛型具体化的常见方式。

`class SpecialSet extends AbstractSet<String> {...}`

SpecialSet 类使用参数化类型 String 扩展了 AbstractSet。

`class SpecialSet<E,P> extends AbstractSet<E> {...}`

SpecialSet 类使用参数类型 E 扩展了 AbstractSet。类型 P 是 SpecialSet 类特有的。

`class SpecialSet<E> extends AbstractSet {...}`

SpecialSet 类是一个泛型类，它应该参数化 AbstractSet 类的泛型类型。因为这里扩展了 AbstractSet 类的原始类型（而不是泛型），这里不会发生参数化。当试图进行方法调用时，会生成编译警告。

```
class SpecialSet extends AbstractSet {...}
```

SpecialSet 类扩展了原始类型的 AbstractSet 类。因为预
169 期的是泛型版本的 AbstractSet 类，所以当试图进行方法调用时，会生成编译警告。

非泛型类型中的泛型方法

非泛型类型或者原始类型中的静态方法、非静态方法和构造器可以声明为泛型。这里所谓的原始类型的类是相对于泛型类来说的，也就是非泛型的类。

对于非泛型类的泛型方法来说，方法的返回值类型前面必须要带有泛型类型参数（例如 <E>）。但是，类型参数和返回类型之间并没有功能性的关联，除非返回类型也是相同的泛型类型：

```
public class SpecialQueue {
  public static <E> boolean add(E e) {...}
  public static <E> E peek() {...}
}
```

当调用泛型方法时，泛型类型参数要放到方法名前面。在这里，<String> 用来指定泛型类型参数：

```
SpecialQueue.<String>add("White Carnation");
```

第 17 章

Java脚本API

Java 脚本 API（Java Scripting API）是在 Java SE 6 引入的，它允许 Java 应用与脚本语言通过标准接口进行交互。这个 API 在 JSR 223“Scripting for the Java Platform”进行了详细描述，并且包含在了 java.scripting 模块的 javax.script 包中。

脚本语言

目前，已经有多个符合 JSR 223 的脚本引擎实现可供使用。参见附录 B 中“兼容 JSR 223 的脚本语言”以了解这些所支持语言的子集。

脚本引擎实现

`ScriptEngine` 接口为该 API 提供了基础方法。`ScriptEngineManager` 类与这个接口联合起来，提供了搭建脚本引擎的方法。

嵌入脚本到 Java 中

脚本 API 提供了将脚本和 / 或脚本组件嵌入到 Java 应用中的能力。

如下的样例展现了将脚本组件嵌入到 Java 应用中的两种方式：（1）脚本引擎的 eval 方法直接读取脚本语言语法；（2）脚本引擎的 eval 方法读取文件中的语法：

```
import java.io.FileReader;
import java.nio.file.Path;
import java.nio.file.Paths;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class HelloWorld {
  public static void main(String[] args) throws
      Exception {
    ScriptEngineManager m
        = new ScriptEngineManager();
    // 搭建 Nashorn JavaScript 引擎
    ScriptEngine e = m.getEngineByExtension( “js” );
    // Nashorn JavaScript 语法
    e.eval( “print ( ‘Hello, ‘)” );
    // world.js 的内容：print( ‘World!\n’ );
    Path p1 = Paths.get( “/opt/jpg2/world.js” );
    e.eval(new FileReader(p1.toString()));
  }
}

$ Hello, World!
```

调用脚本语言的方法

实现了可选 Invocable 接口的脚本引擎能够提供调用（执行）脚本语言方法的功能，这些方法需要引擎已经进行了解析。

173 如下基于 Java 的 invokeFunction() 方法会调用已解析的 Nashorn 脚本语言函数 greet()：

```
ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine e = m.getEngineByExtension( “js” );
e.eval( “function greet(message) {print(message)}” );
Invocable i = (Invocable) e;
i.invokeFunction( “greet” , “Greetings from Mars!” );

$ Greetings from Mars!
```

从脚本中访问和控制 Java 资源

Java 脚本 API 提供了在已解析的脚本语言代码中访问和控制 Java 资源(对象)的能力。脚本引擎会使用 key-value 绑定的机制。

在这里，已解析的 Nashorn JavaScript 使用 nameKey 进行绑定，在已解析的脚本语言中读取（并打印）一个 Java 数据成员：

```
ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine e = m.getEngineByExtension( “js” );
e.put( “nameKey” , “Gliese 581 c” );
e.eval( “var w = nameKey” );
e.eval( “print(w)” );

$ Gliese 581 c
```

通过使用 key-value 绑定，我们还可以在已解析的脚本语言中修改 Java 数据成员：

```
ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine e = m.getEngineByExtension( “js” );
List<String> worldList = new ArrayList<>();
worldList.add ( “Earth” );
worldList.add ( “Mars” );
e.put( “nameKey” , worldList);
e.eval( “var w = nameKey.toArray();” );
e.eval( “ nameKey.add (\” Gliese 581 c\” )” );
System.out.println(worldList);

$ [Earth, Gliese 581 c]
```

174

搭建脚本语言和引擎环境

在使用脚本 API 之前，我们必须获取并搭建所需的脚本引擎实现。很多的脚本语言在其分发包中包含了 JSR-223 脚本引擎，要么位于单独的 JAR 包之中，要么像 JRuby 那样放到主 JAR 文件中。

搭建脚本语言的环境

如下是搭建脚本语言的步骤：

1．在系统中搭建脚本语言环境，附录 B“兼容 JSR 223 的脚本

语言”一节中包含了一些所支持的脚本语言的下载站点。按照相关的安装指南进行环境搭建。

2．调用脚本的解释器确保其功能正常。通常这些脚本语言会有一个命令行的解释器和图形化的用户界面。

举例来说，对于 JRuby，可以使用如下的命令来校验环境安装的正确性：

```
jruby [file.rb] // 命令行文件
jruby.bat // Windows 批处理文件
```

搭建引擎

如下是搭建脚本引擎的步骤：

1．确定脚本语言分发包中是否包含 JSR-223 脚本 API 引擎。如果已经包含，那第 2 步和第 3 步就不需要了。

175 2．从引擎的 Web 站点上下载脚本引擎文件。

3．将下载的文件放到一个目录中并解压以暴露必要的 JAR。需要注意的是，可选软件（optional software, opt）目录通常会用作安装目录。

提示

要在 Windows 机器上安装和配置特定的脚本语言，我们可能需要一个最小化的兼容 POSIX 的 shell 环境，比如 MSYS 或 Cygwin。

脚本引擎的校验

要校验脚本引擎的安装情况，我们可以编译和 / 或解释脚本语言库和脚本引擎库。如下展示了较早版本的 JRuby，当时它的引擎是外部安装的：

```
javac -cp c:\opt\jruby-1.0\lib\jruby.jar;c:\opt\
jruby-engine.jar;. Engines
```

我们还可以使用简短的程序执行额外的测试。如下的应用会生成一个列表，其中包含了可用的脚本引擎的名称、语言版本以及扩展。注意这个升级版本的 JRuby 在其主 JAR 包中包含了对 JSR-223 的支持，因此，引擎不需要在类路径中单独调用：

```
$ java -cp c:\opt\jruby-9.1.6.0\lib\jruby.jar;.
  EngineReport

import java.util.List;
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngineFactory;

public class EngineReport {                                    176
  public static void main(String[] args) {
    ScriptEngineManager m =
        new ScriptEngineManager();
    List<ScriptEngineFactory> s =
        m.getEngineFactories();
    // 迭代工厂列表
    for (ScriptEngineFactory f: s) {
      // 发布名和版本
      String en = f.getEngineName();
      String ev = f.getEngineVersion();
      System.out.println( "Engine: "
        + en + " " + ev);
      // 语言和版本
      String ln = f.getLanguageName();
      String lv = f.getLanguageVersion();
      System.out.println( "Language: "
        + ln + " " + lv);
      // 扩展
      List<String> l = f.getExtensions();
      for (String x: l) {
        System.out.println( "Extensions: " + x);
      }
    }
  }
}

$ Engine: JSR 223 JRuby Engine 9.1.6.0
$ Language: ruby jruby 9.1.6.0
$ Extensions: rb

$ Engine: Oracle Nashorn 9-ea
$ Language: ECMAScript ECMA - 262 Edition 5.1
$ Extensions: js
```

177

提示

从 Java SE 8 之后，Nashorn JavaScript 作为脚本 API 打包了进来。命令和参数 jjs –scripting 能够调用 Nashorn 引擎，并具有 shell 脚本的特性。

第 18 章

日期和时间API

179

日期和时间 API（Date and Time API，JSR 310）提供了对日期、时间以及日历计算的支持。该 JSR 的参考实现（RI）是 ThreeTen 项目（*http://www.threeten.org/*）并包含在了 JDK 1.8 中。日期和时间 API 位于 java.time package 包及其子包 java.time.chrono、java.time.format、java.time.temporal 和 java.time.zone 中。

JSR 310 实现了如下的设计目标：

- Fluent API，易于读取的链式方法；
- 借助不可变的值类实现线程安全的设计；
- 可扩展的 API 以及日历系统、调节器（adjuster）和查询；
- 可预期的行为，每个方法的行为都是清晰且定义良好的。

日期和时间 API 使用了国际标准化组织（International Organization for Standardization）的日期和时间数据交换模型（ISO 8601）。ISO 8601 标准的正式叫法是“数据元素和交换形式 · 信息交换 · 日期和时间的表示方法”。这个标准以公历（Gregorian）为基础，也支持区域性的日历。

参见附录 A 了解 Fluent API 的更多信息。

180

与遗留系统的互操作

JSR 310 取代但并没有废弃 java.util.Date、java.util.Calendar、java.util.DateFormat、java.util.GregorianCalendar、java.util.TimeZone 和 java.sql.Date。JDK 8 提供了一些方法，实现这些类与 JSR 310 类型之间的相互转换：

```
// 遗留的 Calendar -> 新的 Instant-> 遗留的 Date
Calendar c = Calendar.getInstance();
Instant i =  c.toInstant();
Date d = Date.from(i);

/*
 * 新的 ZonedDateTime -> 遗留的 GregorianCalendar  -> 新的 LocalDateTime
ZonedDateTime zdt =
   ZonedDateTime.parse( “2014-02-24T11:17:00+01:00”
   + “[Europe/Gibraltar]” )
GregorianCalendar gc = GregorianCalendar.from(zdt);
LocalDateTime ldt
  = gc.toZonedDateTime().toLocalDateTime();
```

区域性日历

JSR 310 允许使用额外的新日历。当创建新的日历时，需要提供实现 Era、Chronology 和 ChronoLocalDate 接口的类。

该 API 已经打包了 4 个区域性的日历：

- Hijrah
- 日本历
- 民国历
- 泰国佛教历

181 在区域性日历中，所使用的不是 ISO 日历的主类。

ISO 日历

API 的 java.time 包中提供了 ISO 8601 日历系统，它是基于公历的。API 中的这个包以及相关的包提供了易于使用的接口，在

下面的例子中可以看到根据两个日期确定年龄差的方法。这个样例改编自 Gliesians 年龄差计算器：

```
final String YANNI_BIRTH_YEAR = “1954”;
final String ADELE_BIRTH_YEAR = “1988”;

Year birthYear1 = Year.parse(YANNI_BIRTH_YEAR);
Year birthYear2 = Year.parse(ADELE_BIRTH_YEAR);
long diff
  = ChronoUnit.YEARS.between(birthYear1,
                             birthYear2);
System.out.println( “There is an age difference of “
  + Math.abs(diff) + “ years.” );

 $ There is an age difference of 34 years.
```

下面列出了该 API 主要的类，主要的介绍文字来源于在线 API：

`Instant`

时间线上的一个瞬时点，基于标准 Java 的时间起点 1970-01-01T00:00:00Z 进行测算。

`LocalDate`

代表日期的不可变的日期 - 时间对象，展现为年 - 月 - 日。

`LocalTime`

代表时间的不可变的日期 - 时间对象，展现为小时 - 分 - 秒。

`LocalDateTime`

代表日期和时间的不可变的日期 - 时间对象，展现为年 - 月 - 日 - 小时 - 分 - 秒。 182

`OffsetTime`

代表时间的不可变的日期 - 时间对象，展现为小时 - 分 - 秒 - 偏移。

`OffsetDateTime`

带有偏移的不可变日期 - 时间对象。存储所有的日期和时间字段，精确到纳秒，同时还包括从 UTC/ 格林威治（Greenwich）计算的偏移。

`ZonedDateTime`

带有时区的不可变日期 - 时间对象。存储所有的日期和时间字段，精确到纳秒，同时还包括区域的偏移，用来处理不确定的本地日期 - 时间。

`ZoneOffset`

时区的偏移。某个时区相对于格林威治 /UTC 的时间差。

`ZonedId`

时区标识。用来识别在 `Instant` 和 `LocalDateTime` 之间转换的规则。

`Year`

不可变的日期 - 时间对象，代表了年份。

`YearMonth`

不可变的日期 - 时间对象，代表了年份和月份。

`MonthDay`

不可变的日期 - 时间对象，代表了月份和日期。

`DayOfWeek`

每周中各个日期的枚举：Monday、Tuesday、Wednesday、Thursday、Friday、Saturday 和 Sunday。

183 `Month`

一年中各个月份的枚举：January、February、March、April、May、June、July、August、September、October、November 和 December。

`Duration`

基于时间的时间量，以秒进行计算。

`Period`

基于日期的时间量。

`Clock`

Clock 提供了使用时区访问当前 Instant、日期和时间的功能。对它的使用是可选的。

接下来将会重点讲述其中某些类的关键属性和用法。

机器接口

JSR 310 使用 UNIX Epoch 作为默认的 ISO 8301 日历，它的零起始点从 1970-01-01T00:00Z 开始。从这个点开始，时间会持续增加，实例的负数值用来代表在它之前的时间。

要获取当前时间的实例，只需简单地调用 `Instant.now()` 方法即可：

```
Instant i = Instant.now();

System.out.println( "Machine: " + i.toEpochMilli());
$ Machine: 1478860514417

System.out.println( "Human: " + i);
$ Human: 2016-11-11T10:35:31.727Z
Clock 提供了使用时区访问当前 Instant、日期和时间的功能：
Clock clock1 = Clock.systemUTC();
Instant i1 = Instant.now(clock1);

ZoneId zid = ZoneId.of( "Europe/Vienna" );                    184
Clock clock2 = Clock.system(zid);
Instant i2 = Instant.now(clock2);
```

日期 - 时间 API 使用了时区数据库（Time Zone Database，TZDB，*http://www.iana.org/time-zones*）。

Duration 与 Period

Duration 是一个基于时间的量，包含了天数、小时数、分钟数、秒数以及纳秒数。Duration 记录了两个实例在时间线上的持续时间。

Duration 的用法是可解析的字符串 PnDTnHnMnS 形式，其中 P 代表了这是一个阶段，T 代表了时间。D、H、M 和 S 分别代表

的是天数、小时数、分钟数以及秒数，在单位前面是实际的值（n）：

```
Duration d1 = Duration.parse( “P2DT3H4M1.1S” );
```

Duration 也可以使用 `of[Type]` 方法来创建。小时、分钟、秒以及毫秒都可以在相关的状态位置上进行加法计算：

```
Duration d2 = Duration.of(41, ChronoUnit.YEARS);

Duration d3 = Duration.ofDays(8);
d3 = d3.plusHours(3);
d3 = d3.plusMinutes(30);
d3 = d3.plusSeconds(55).minusNanos(300);
```

`Duration.between()` 可以基于一个起始时间和结束时间创建 `Duration`：

```
Instant birth = Instant.parse( “1967-09-
15T10:30:00Z” );
Instant current = Instant.now();
Duration d4 = Duration.between(birth, current);
System.out.print( “Days alive: “ + d4.toDays());
```

Period 是一个基于日期的时间量，它包含了年数、月数和天数。

185 Period 的用法是可解析的字符串 PnYnMnD 形式，其中 P 代表了这是一个阶段。Y、M 和 D 分别代表的是年数、月数和天数，在单位前面是实际的值（n）：

```
Period p1 = Period.parse( “P10Y5M2D” );
```

Period 也可以使用 `of[Type]` 方法来创建。年数、月数和天数都可以在相关的状态位置上进行加法或减法计算：

```
Period p2 = Period.of(5, 10, 40);
p2 = p2.plusYears(100);
p2 = p2.plusMonths(5).minusDays(30);
```

JDBC 与 XSD 映射

java.time 和 java.sql 类型之间的互相转换已经实现了。表 18-1 展现了 JSR 310 类型与 SQL、XML 模式（XSD）类型之间的映射关系。

表18-1　JDBC与XSD映射

JSR 310 类型	SQL 类型	XSD 类型
LocalDate	DATE	xs:time
LocalTime	TIME	xs:time
LocalDateTime	TIMESTAMP WITHOUT TIMEZONE	xs:dateTime
OffsetTime	TIME WITH TIMEZONE	xs:time
OffsetDateTime	TIMESTAMP WITH TIMEZONE	xs:dateTime
Period	INTERVAL	.

格式化

186

DateTimeFormatter 类提供了格式化功能，可以打印和解析日期 - 时间对象。下面的样例阐述了如何使用该类的 ofPattern() 方法和模式字符。可用的模式字符在 DateTimeFormatter 类的 Javadoc 中进行了标识：

```
LocalDateTime input = LocalDateTime.now();
DateTimeFormatter format
  = DateTimeFormatter.ofPattern( “yyyyMMddhhmmss” );
String date = input.format(format);
String logFile = “simple-log-” + date + “.txt” ;
```

表 18-2 列出了预先定义好的格式化器，可以按照如下的结构来使用：

```
System.out.print(LocalDateTime.now()
  .format(DateTimeFormatter.BASIC_ISO_DATE));
```

表18-2　预先定义的格式化器

类	格式	样例
LocalDateTime	BASIC_ISO_DATE	20140215
LocalDateTime	ISO_LOCAL_DATE	2014-02-15
OffsetDateTime	ISO_OFFSET_DATE	2014-02-15-05:00
LocalDateTime	ISO_DATE	2014-02-15
OffsetDateTime	ISO_DATE	2014-02-15-05:00

类	格式	样例
LocalDateTime	ISO_LOCAL_TIME	23:39:07.884
OffsetTime	ISO_OFFSET_TIME	23:39:07.888-05:00
LocalDateTime	ISO_TIME	23:39:07.888
OffsetDateTime	ISO_TIME	23:39:07.888-05:00
LocalDateTime	ISO_LOCAL_DATE_TIME	2014-02-15T23:39:07.888
OffsetDateTime	ISO_OFFSET_DATE_TIME	2014-02-15T23:39:07.888-05:00
ZonedDateTime	ISO_ZONED_DATE_TIME	2014-02-15T23:39:07.89-05:00[America/New_York]
LocalDateTime	ISO_DATE_TIME	2014-02-15T23:39:07.891
ZonedDateTime	ISO_DATE_TIME	2014-02-15T23:39:07.891-05:00[America/New_York]
LocalDateTime	ISO_ORDINAL_DATE	2014-046
LocalDate	ISO_WEEK_DATE	2014-W07-6
ZonedDateTime	RFC_1123_DATE_TIME	Sat, 15 Feb 2014 23:39:07 -0500

187

第 19 章

Lambda表达式 189

Lambda 表达式（Lambda expression，λE），也被称为闭包（closure），提供了一种方式来表示匿名方法。λE 由 Lambda 项目（*http://openjdk.java.net/projects/lambda/*）提供支持，它允许创建和使用单方法的类。这些方法有一个基本语法，能够省略修饰符、返回类型和可选参数。λE 规范是在 JSR 335 中定义的，它分成了 7 个部分：函数式接口、lambda 表达式、方法和构造器引用、poly 表达式、类型化与执行（typing and evaluation）、类型推断以及默认方法。本章主要关注前两项。

λE 基础

λE 必须有一个函数式接口（functional interface，FI）。FI 是具有一个抽象方法以及零个或多个默认方法的接口。FI 为 lambda 表达式和方法引用提供了目标类型，在理想情况下，它应该添加上 `@FunctionalInterface` 注解，以帮助开发人员和编译器理解设计意图，如下面的样例代码所示：

```
@FunctionalInterface
public interface Comparator<T> {
  // 只允许有一个抽象方法
  int compare(T o1, T o2);                                    190
  // 允许覆盖
```

```
    boolean equals(Object obj);
    // 允许存在可选的默认方法
}
```

λ E 语法和样例

Lambda 表达式一般会包括一个参数列表、一个返回类型和方法体：

```
(parameter list) -> { statements; }
```

λ E 的样例如下所示：

```
() -> 66
(x,y) -> x + y
(Integer x, Integer y) -> x*y
(String s) -> { System.out.println(s); }
```

在下面简单的 JavaFX GUI 应用中，当按钮点击的时候，将会添加文本到标题栏上。代码中用到了 `EventHandler` 函数式接口，该接口具有一个抽象方法 `handle()`：

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
public class JavaFxApp extends Application {
  @Override
  public void start(Stage stage) {
    Button b = new Button();
    b.setText( “Press Button to Set Title” );
    // 匿名内部类
    b.setOnAction(new EventHandler<ActionEvent>() {
      @Override
      public void handle(ActionEvent event) {
        stage.setTitle( “λEs rock!” );
      }
    });
    StackPane root = new StackPane();
    root.getChildren().add(b);
    Scene scene = new Scene(root, 300, 100);
    stage.setScene(scene);
    stage.show();
  }
```

191

```
  public static void main(String[] args) {
    launch();
  }
}
```

要将这个匿名内部类重构为 lambda 表达式，参数类型需要是 (ActionEvent event) 或 (event)，所需的功能要作为语句放到内容体中：

```
// 使用 lambda 表达式
b.setOnAction((ActionEvent event) -> {
  stage.setTitle( "λEs rock!" );
});
```

提示

现代的 IDE 具有将匿名内部类转换为 lambda 表达式的特性。

参见第 15 章“Comparator 函数式接口”中获取 lambda 表达式的另外一个样例，该样例用到了 Comparator 函数式接口。

方法与构造器引用

方法引用会指向一个已有的方法，而不会调用它，其类型包括静态方法引用、特定对象的实例方法、特定对象的 super 方法以及特定类型的任意对象的实例方法。方法引用是执行某个方法的 lambda 表达式，如下面的样例所示：

192

```
"some text"::length  // 获取 String 的长度
String::length // 获取 String 的长度
CheckAcct::compareByBalance  // 静态方法的引用
myComparator::compareByName // 特定对象的实例方法
super::toString // 特定对象的 super 方法
String::compareToIgnoreCase // 任意对象的实例方法
ArrayList<String>::new  // 新 ArrayList 的构造器
Arrays::sort  // 排序数组元素
```

特定用途的函数式接口

使用注解标注的 FI 如表 19-1 所示，它们的用途在于满足所在包/API 的相关需求，Java SE API 中的函数式接口并没有全部添加注解：

表 19-1 特定用途的 FI

API	类	方法
AWT	KeyEventDispacter	dispatchKeyEvent (KeyEvent e)
AWT	KeyEventPostProcessor	postProcessKeyEvent (KeyEvent e)
IO	FileFilter	accept(File pathname)
IO	FilenameFilter	accept(File dir, String name)
LANG	Runnable	run ()
Nashorn	DiagnosticListener	report (Diagnostic diagnostic)
NIO	DirectorStream	iterator ()
NIO	PathMatcher	matches (Path path)
TIME	TemporalAdjuster	adjustInto (Temporal temporal)
TIME	TemporalQuery	queryFrom (TemporalAccessor temporal)
UTIL	Comparator	compare (T o1, T o2)
CONC	Callable	call ()
LOG	Filter	isLoggable (LogRecord record)
PREF	PreferenceChangeListener	preferenceChange (PreferenceChangeEvent evt)

通用的函数式接口

在 java.util.function 包中包含了通用的 FI，主要用于 JDK 特性的实现。表 19-2 列出了这些 FI。

表19-2　函数式接口

类	方法
Consumer	accept (T t)
BiConsumer	accept (T t, U u)
ObjDoubleConsumer	accept (T t, double value)
ObjIntConsumer	accept (T t, int value)
ObjLongConsumer	accept (T t, long value)
DoubleConsumer	accept (double value)
IntConsumer	accept (int value)
LongConsumer	accept (long value)
Function	apply (T t)
BiFunction	apply (T t, U u)
DoubleFunction	apply (double value)
IntFunction	apply (int value)
LongFunction	apply (long value)
BinaryOperator	apply (Object, Object)
ToDoubleBiFunction	applyAsDouble (T t, U u)
ToDoubleFunction	applyAsDouble (T value)
IntToDoubleFunction	applyAsDouble (int value)
LongToDoubleFunction	applyAsDouble(long value)
DoubleBinaryOperator	applyAsDouble (double left, double right)
ToIntBiFunction	applyAsInt(T t, U u)
ToIntFunction	applyAsInt(T value)
LongToIntFunction	applyAsInt(long value)
DoubleToIntFunction	applyAsInt(double value)
IntBinaryOperator	applyAsInt (int left, int right)

194

类	方法
ToLongBiFunction	applyAsLong (T t, U u)
ToLongFunction	applyAsLong (T value)
DoubleToLongFunction	applyAsLong (double value)
IntToLongFunction	applyAsLong (int value)
LongBinaryOperator	applyAsLong (long left, long right)
BiPredicate	test (T t, U u)
Predicate	test (T t)
DoublePredicate	test (double value)
IntPredicate	test (int value)
LongPredicate	test (long value)
Supplier	get()
BooleanSupplier	getAsBoolean()
DoubleSupplier	getAsDouble()
IntSupplier	getAsInt()
LongSupplier	getAsLong()
UnaryOperator	identity()
DoubleUnaryOperator	identity()
IntUnaryOperator	applyAsInt (int operand)
LongUnaryOperator	applyAsInt (long value)

195

关于 λE 的资源

本节介绍了 λE 相关教程和社区资源的链接。

教程

目前，有一些由 Oracle、O’Reilly Learning 和 Maurice Naftalin 提供的综合教程。

- Java 教程：“Lambda 表达式”（*https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html*）。
- O’Reilly Learning 上的“Java 8 函数式接口”（*https://www.*

oreilly.com/learning/java-8-functional-interfaces）。

- Maurice Naftalin 的 Lambda FAQ：“Your questions answered: all about Lambdas and friends”（*http://www.lambdafaq.org/*）。

社区资源

在线论坛、邮件列表和教学视频提供了学习和使用 λE 的资料：

- CodeRanch 上的 Java 8 新特性（如 λE）论坛（*https://coderanch.com/f/143/java*）；
- 在 YouTube 上的 Oracle Learning Library（*https://www.youtube.com/watch?v=Oif7Udc5ImM&list=PLKCk3OyNwIzv6qi-LuJkQ0tGjF7gZTpqo*）。

第 20 章

JShell：Java Shell

JShell 最初被称为 Kulla 项目，它是一个交互式命令行读取 - 执行 - 打印 - 循环（read-eval-print-loop ，REPL）工具，它是在 Java 9 SDK 引入的。与 Python 的 ipython 和 Haskell 的 ghci 解释器类似，JShell 允许用户实时执行和测试代码片段，能够避免创建测试项目或测试类来包含 `main` 函数的麻烦。

本章中的代码基于 JShell 9-ea 版本进行了测试。

起步

JShell 可以通过 NetBeans IDE 的菜单来启动（Tools → Java Platform Shell），在 Windows 下可以从 JDK 安装路径的 */bin/* 目录下通过运行 jshell.exe 命令行来启动，在 POSIX 环境下可以通过 jshell 命令来启动。

环境加载完成后，将会看到如下的欢迎提示：

```
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro


jshell>
```

从这里，我们就可以输入、执行或修改代码片段，或者通过内

置的命令与 JShell 环境进行交互。

片段

JShell 的操作都是基于*代码片段*（snippet）运行的，它指的是用户通过 `jshell>` 提示符输入的代码段。每个片段的形式都要符合 JLS（Java 语言规范）的定义，表 20-1 对其进行了简要总结：

表20-1　所允许的片段形式

Java 语言规范定义的形式	样例
主表达式（Primary）	`10 / 2`
语句（Statement）	`if (value == null) {` `numWidgets = 0; }`
类声明 （ClassDeclaration）	`class Foo { }`
方法声明 （MethodDeclaration）	`void sayHello () { System.` `out.println( “Hello” ); }`
字段声明 （FieldDeclaration）	`boolean isAnchovyLover =` `true;`
接口声明 （InterfaceDeclaration）	`interface eventHandler {` `void onThisEvent(); }`
导入声明 （ImportDeclaration）	`import java.math.` `BigInteger;`

修饰符

JShell 在处理修饰符方面与标准编译的 Java 是不同的。其中最重要的一点在于，它在顶层声明（也就是，在主 JShell “沙箱”之内以及类 / 方法声明的作用域或其他内嵌上下文之外）中限制了几个修饰符的使用。如下的样例将会警告用户正在尝试使用不恰当的 `private` 修饰符：

```
jshell> private double airPressure
|  Warning:
|  Modifier ‘private’  not permitted in top-level
```

199

```
   declarations, ignored
|  private double airPressure;
|  ^-----^
airPressure ==> 0.0

jshell> class AirData { private double airPressure; }
|  created class AirData
```

表 20-2 概要总结了 JShell 的修饰符策略。

表20-2 JShell修饰符规则

修饰符	规则
private、protected、public、final、static	在顶层声明中将会给出警告并忽略
abstract、default	只能用到类声明中
default、synchronized	在顶层声明中禁止使用

流控制语句

类似的，流控制语句 break、continue 以及 return 在顶层中也不允许使用，因为它在这样的上下文中没有相应的含义。

包声明

在 JShell 中不允许进行包声明，因为 JShell 代码都放到了一个暂时的包 transient 之中。

使用 JShell

正如在“起步”一节所提到的，我们与 JShell 的交互主要包含输入、操作以及执行代码片段。如下内容详细介绍了如何使用各种主要的代码片段，以及保存和加载代码与输入历史、恢复和持久化 JShell 状态。

主表达式

200

JShell 将会立即评估和 / 或执行在提示符中输入的主表达式（Primary Expression）：

```
jshell> 256 / 8
$1 ==> 32

jshell> true || false
$2 ==> true

jshell> 97 % 2
$3 ==> 1

jshell> System.out.println( “Hello, Dave. Shall we
continue the game?” )
Hello, Dave. Shall we continue the game?

jshell> StringBuilder sb = new StringBuilder( “HAL” )
sb ==> HAL

jshell> sb.append( “ 9000” )
$4 ==> HAL 9000
```

注意，JShell 会自动在表达式和语句的结尾处添加缺失的分号。但是，在代码块中声明方法、类或其他代码时，分号依然是必须要添加的。

依赖

命令“/imports”将会返回目前导入到工作空间的库列表：

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

201

上面的结果展现了 JShell 为每个新的工作空间默认导入的库。其他的库可以通过 import 命令进行导入：

```
jshell> import java.lang.StringBuilder
```

表达式和代码块

与主表达式类似，代表语句的代码片段也会在输入后立即执行：

```
jshell> double[] tempKelvin = {373.16, 200.19, 0.0}
tempKelvin ==> double[3] { 373.16, 200.19, 0.0 }
```

当语句包含一个或多个代码块时，JShell 提示符在第一次按下回车键之后变成了新行提示符（...>）并且会继续逐行读入代码片段，直到顶层的代码块结束为止：

```
jshell> import java.text.DecimalFormat

jshell> DecimalFormat df = new DecimalFormat( “#.#” );
df ==> java.text.DecimalFormat@674dc

jshell> double[] tempFahrenheit = {30.8, 77.0, 29.3,
60.2 }
tempFahrenheit ==> double[5] { 30.8, 77.0, 29.3, 60.2
}

jshell> for (double temp : tempFahrenheit) {
   ...> double tempCelsius = ((temp - 32)*5/9);
   ...> System.out.println(temp + “ degrees F is equal
to “
   ...> + df.format(tempCelsius) + “ degrees C.\n” );
   ...> }
30.8 degrees F is equal to -0.7 degrees C.
77.0 degrees F is equal to 25 degrees C.
29.3 degrees F is equal to -1.5 degrees C.
60.2 degrees F is equal to 15.7 degrees C.
```

202

如果某个代码块包含了编译错误，比如语法错误，片段既不会创建也不会执行，必须重新输入。尽管可以在提示符中通过上翻箭头在缓冲中查阅之前的命令，但这依然可能是一个非常乏味的过程。在使用命令行时，输入大块的代码要非常小心。

命令“/!”能够用来重新执行刚刚执行过的代码片段。类似的，“/-<n>”能够根据所提供的数字执行之前的代码片段：

```
jshell> System.out.println( “Hello” );
Hello
jshell> System.out.println( “World” );
World
jshell> /!
```

```
System.out.println( “World” );
World
jshell> /-3
System.out.println( “Hello” );
Hello
```

方法和类声明

在 JShell 中声明方法的方式与其他的语句和代码块是相同的，可以通过命令行进行调用：

```
jshell> double KELVIN = 273.16
KELVIN ==> 273.16

jshell> double DRY_AIR_GAS_CONSTANT = 287.058
DRY_AIR_GAS_CONSTANT ==> 287.058

jshell> double getDryAirDensity(double temperature,
   ...>    double atmosphericPressure) {
   ...> // convert from hPa to Pa
   ...> double airDensity = atmosphericPressure * 100
   ...>   / (DRY_AIR_GAS_CONSTANT
   ...>   * (temperature + KELVIN));
   ...> return airDensity;
   ...> }
|  created method getDryAirDensity(double,double)

jshell> double todaysAirDensity =
   ...> getDryAirDensity(15, 1013.25)
todaysAirDensity ==> 1.2249356158607942
```

203

“/methods”命令将会返回当前工作空间中已有的方法列表，并且包含它们的签名：

```
jshell> /methods
|    double getDryAirDensity (double,double)
```

声明类的方式是相同的。在下面的样例中，我们在一个工具类中封装了空气密度计算器的代码，并且在类中使用了 `static final` 修饰符来声明我们的常量：

```
jshell> class AirDensityUtils {
  ...> private static final double KELVIN = 273.16;
  ...> private static final double
  ...>   DRY_AIR_GAS_CONSTANT = 287.058;
  ...>
```

```
  ...> double getDryAirDensity(double temperature,
  ...>   double atmosphericPressure) {
  ...> // convert from hPa to Pa
  ...> double airDensity = atmosphericPressure * 100
  ...> / (DRY_AIR_GAS_CONSTANT
  ...> * (temperature + KELVIN));
  ...> return airDensity;
  ...> }
  ...> }
|  created class AirDensityUtils
```

类的方法和成员可以在命令行中通过标准的 Java 点号操作符来进行访问。尽管 `AirDensityUtils` 是一个工具类，但是它不能 204 在静态上下文中进行访问，因为 `static` 操作符不允许在顶层声明中使用，所以它必须要实例化：

```
jshell> new AirDensityUtils().
  ...>    getDryAirDensity(15, 1013.25)
$5 ==> 1.2249356158607942
```

其他的类型，比如接口和枚举，也采用相同的方式来声明。“/types”命令将会返回当前工作空间中所有类型的列表：

```
jshell> interface EventHandler { void
onWeatherDataReceived(); }
|  created interface EventHandler

jshell> enum WeatherCondition { RAIN, SNOW, HAIL }
|  created enum WeatherCondition

jshell> /types
|    class AirDenstityUtils
|    interface EventHandler
|    enum WeatherCondition
```

查看、删除和修改片段

在定义之后，代码片段可以很容易地进行查看、删除和修改。“/list”命令将会展现所有代码片段的列表，并且会展示对应的识别数字：

```
jshell> /list

   1 : double KELVIN = 273.16;
   2 : double DRY_AIR_GAS_CONSTANT = 287.058;
```

```
3 : double getDryAirDensity(double temperature,
      double atmPressure) {
      // convert from hPa to Pa
      double airDensity = atmoPressure * 100
      / (DRY_AIR_GAS_CONSTANT
205>  * (temperature + KELVIN));
    return airDensity;
    }
4 : class AirDensityUtils {
    private static final double
      KELVIN = 273.16;
    private static final double
      DRY_AIR_GAS_CONSTANT = 287.058;

    double getDryAirDensity(double temperature,
      double atmPressure) {
      // convert from hPa to Pa
      double airDensity = atmPressure * 100
      / (DRY_AIR_GAS_CONSTANT *
      (temperature + KELVIN));
      return airDensity;
    }
    }
```

在JShell命令中，代码片段可以通过名字或识别数字进行引用。在前面的样例中，我们定义了两个多余的伪常量DRY_AIR_GAS_CONSTANT和KELVIN，因为在类中封装了它们和getDryAirDensity(double, double)方法，所以我们使用“/drop”命令将它们删除掉：

```
jshell> /drop KELVIN
|  dropped variable KELVIN

jshell> /drop 2
|  dropped variable DRY_AIR_GAS_CONSTANT
```

修改和替换之前定义的代码片段也非常容易。执行该操作的方法将会覆盖掉原有的方法：

```
jshell> double getDryAirDensity(double temperature,
   ...>   double atmPressure) {
   ...>   // We don't need this method anymore,
   ...>   // but let's replace it anyway!
   ...> }
206>|  replaced method getDryAirDensity(double,double)
```

如果涉及到大量的代码或仅仅进行小范围的调整，这种方法并不实用。所幸的是，JShell 允许通过“/edit <name>”或“/edit <id>”命令在外部编辑器中修改代码片段：

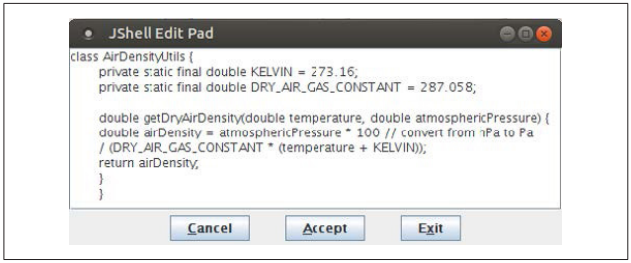


图20-1　JShell默认的编辑面板

在图 20-1 中，我们在 JShell 默认的编辑面板中打开了 `AirDensityUtils` 方法进行编辑。不过，在这个任务中，JShell 所启动的编辑器可以通过“/set editor <_command_>”来指定，其中“<_command_>”是用来启动编辑器的操作系统命令。比如，在 Linux 下就可以使用“/set editor vim”或“/set editor emacs”。

不管你使用那么方式来修改代码片段，引用或依赖此片段的其他片段不会受到变化的影响。

保存、加载和状态

“/save <_file_>”命令会将当前活跃代码片段的源码保存到指定的文件中。使用“-all”标记则会将当前会话中的所有源码保存下来，其中包括被覆盖或被拒绝的代码片段，使用“-history”标记会按照输入的顺序保存所有的代码片段和命令。

与之相对的是，“/open <_file_>”命令则会从一个特定的文件中 207
加载内容作为 JShell 的输入。需要注意，如果这个文件中包含包声明，将无法成功加载。

在会话处于活跃时，JShell 的状态可以重置或恢复。“/reset”命令会重置 JShell 的状态，也就是清除所有输入的代码、重启执

行状态并重新执行启动代码。

“/reload”命令则会重置 JShell 的代码和执行状态并且重放所有合法的片段和命令。重放会从会话启动的地方开始或者从上一次的“/reset”或“/reload”开始，重新执行所有发生过的事情。另外，如果在启动的时候使用“/reload -restore”命令，将会从上一个会话中恢复 JShell 的状态。

JShell 在执行完“/ reset”或“/reload”之后，可以通过“/set start <_file_>”加载代码片段，其中 file 保存了片段代码的集合。另外，使用“/retain start”命令能够让 JShell 在每次启动的时候加载代码。如果在不同的会话之间都要使用相同的方法和类，那么这将是一个很有用的特性。

JShell 的特性

相对于其他的 Shell 脚本和解释器环境，JShell 提供了一些便利的特性，这些特点也让它与传统编译后的 Java 有所不同。在这些特性中，特别值得介绍的是 scratch 变量、Tab 智能自动完成、前向引用（forward referencing）、检查型异常的宽松处理以及对顶层变量的处理。

Scratch 变量

独立主表达式或方法调用的返回值存储在 scratch 变量中，这些变量以美元符号（$）作为前缀并且可以在 JShell 环境中访问：

```
jshell> 21 + 20
$6 ==> 41

jshell> $6 + 1
$7 ==> 42

jshell> “The meaning of life is “ + $7
$8 ==> “The meaning of life is 42”

jshell> $8.getClass().getName()
$9 ==> “java.lang.String”
```

想要查看语句的返回值却不想调用 getClass()，可以将 JShell 的 feedback 模型设置为 verbose：

```
jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> 7.0 % 2
$10 ==> 1.0
|  created scratch variable $10 : double
```

Tab 自动补全

JShell 包含了现代命令行解释器和 shell 最便利的一项特性，那就是 Tab 自动补全。当用户单击 Tab 键的时候，JShell 会自动补全可能要输入的类型变量、片段或对象名。

如果存在不确定的情况，JShell 会为用户提供一个可选项的列表。在下面的例子中，用户在输入 temp 之后，单击了 Tab 键，但是在当前的环境中有 3 个变量以这些字符开头：

```
shell> temp
tempCelsius      tempFahrenheit   tempKelvin
```

前向引用

209

JShell 允许在代码片段的声明中使用前向引用（forward referencing）功能。也就是，定义的方法可以引用尚未定义的其他方法、类或变量。但是，这些未定义的条目必须要在方法调用或引用之前定义：

```
jshell> void getDryAirDensity(
   ...>   MeasurementSystem unit) {
   ...> temperature = x;
   ...> pressure = y;
   ...> adjustUnits(x, y, unit);
   ...> // calculation code
   ...> }
|  created method
getDryAirDensity(MeasurementSystem
), however, it cannot be referenced until
class Measure mentSystem, variable
temperature, variable pressure, and
variable y are declared
```

未定义的类不能用作方法声明的返回类型，也不能引用未定义类的成员、方法或构造器。

检查型异常

如果某个独立的语句调用了抛出检查型异常的方法，JShell 在幕后会自动提供异常处理功能，不需要用户任何额外的输入：

```
jshell> BufferedReader bReader = new BufferedReader(
   ...>   new FileReader( "message.txt" ))
bReader ==> java.io.BufferedReader@1e3c938

jshell> String txtLine;
txtLine ==> null

jshell> while ((txtLine = bReader.readLine()) != null)
   ...> { System.out.println(txtLine); }
I don't like macaroni cheese.  And I don't like
scrambled eggs.  And I don't like cocoa.

jshell> bReader.close()
```

210

在前面的例子中，3 个文件 I/O 操作会成功执行而不需任何的 IOException 处理。但是，如果代码片段是方法或类的声明（也就是说，它不是单独、离散的语句），那么必须要像往常那样处理所有抛出的异常：

```
jshell> void displayMessage() {
   ...> BufferedReader bReader = new BufferedReader(
   ...>   new FileReader( "message.txt" ));
   ...> String txtLine;
   ...> while ((txtLine = bReader.readLine()) != null)
   ...>   { System.out.println(txtLine); }
   ...> bReader.close();
   ...> }
|  Error:
|  unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
|  BufferedReader bReader = new BufferedReader(new
FileReader( "message.txt" ));
|
|  Error:
|  unreported exception java.io.IOException; must be
caught or declared to be thrown
|  while ((textLine = bReader.readLine()) != null)
{ System.out.println(textLine); }
```

```
|
|  Error:
|  unreported exception java.io.IOException; must be
caught or declared to be thrown
|  bReader.close();
```

层级结构和作用域

211

JShell 环境非常有意思的一个特性就是在顶层声明的变量、方法和类能够在 JShell 层级结构的任何作用域中访问。

这是因为 JShell 会将代码片段封装到一个合成类（synthetic class）中，这样 Java 编译器就能理解它们，Java 编译器只能识别顶层上的 import 语句和类声明。

具体来说，顶层的 JShell 变量、方法和类声明会成为合成类的静态成员，语句和主表达式则会封装到合成方法中。导入语句会被识别为顶层构造，原封不动地放到合成类的顶部。

如下样例中的方法和类都会在各自的作用域中读取并修改顶层的 double 变量 pressure：

```
jshell> double pressure = 30.47
pressure ==> 30.47

jshell> void convertPressureinHgTohPa() {
   ...> pressure = pressure * 33.86389;
   ...> }
|  created method convertPressureinHgTohPa()

jshell> convertPressureinHgTohPa()

jshell> pressure
pressure ==> 1031.8327282999999

jshell> class WeatherStation {
   ...> double mAirPressure;
   ...> public WeatherStation() {
   ...> mAirPressure = pressure;
   ...> }
   ...> }
|  created class WeatherStation

jshell> WeatherStation ws = new WeatherStation()
ws ==> WeatherStation@b1ffe6
```

212

```
jshell> ws.mAirPressure
$11 ==> 1031.8327282999999
```

坚持编码最佳实践的人应该不会赞赏这样的例子，但是 JShell 是一个非常好的实验场所，能够进行快速地体验、非正式的代码测试，有些人可能会发现这些特性挺有用的。

JShell 命令小结

表 20-3 列出了 JShell 环境所有可用的命令。在 JShell 环境中，我们可以随时使用“/help”命令查阅。

表20-3　JShell命令

命令	描述
/list [<name or id>|-all|-start]	列出输入到 JShell 的源码
/edit <name or id>	根据名称或 id 编辑源码条目
/drop <name or id>	根据名称或 id 删除源码条目
/save [-all|-history|-start] <file>	保存特定的片段和 / 或命令到文件中
/open <file>	打开文件作为源输入
/vars [<name or id>|-all|-start]	列出已声明的变量和对应的值
/methods [<name or id>|-all|-start]	列出已声明的方法和对应的签名
213 /types [<name or id>|-all|-start]	列出已声明的类、接口和枚举
/imports	列出当前 JShell 活跃的导入
/exit	退出 JShell，不保存
/reset	重置 JShell 的状态
/reload [-restore] [-quiet]	重置 JShell 的状态，并且重放从 JShell 启动或从上一条“/reset”或“/reload”开始的命令

续表

命令	描述
/history	展现从 JShell 启动开始，输入的所有代码片段和命令
/help [<command>|<subject>]	展现 JShell 可用的命令列表，或者特定命令或对象的帮助信息和详细介绍
/set editor|start|feedback|mode|prompt|truncation|format	设置 JShell 的配置项
/retain editor|start|feedback|mode	保留配置信息，以便于在后续的会话中使用
/? [<command>|<subject>]	等价于“/help”
/!	重新运行上一条代码片段
/<id>	根据 ID 重新运行一个代码片段
/-<n>	根据顺序编号重新运行之前的某个代码片段

第 21 章

Java模块系统

Java 9 引入了 Jigsaw 项目，该项目为平台增加了模块化功能，同时模块化了 JDK 本身。Jigsaw 有双重的目标：支持可靠的配置并为 Java 添加强封装。借助模块化功能，我们可以限制哪些包是公开的，并且能够确保当应用启动时，它所要求的依赖都能得到满足。

Java 平台模块化系统（Java Platform Module System，JPMS）作为 JVM 中单独的一个层来实现。这与其他的模块化系统是有所不同的，比如借助类加载机制实现的 OSGi 方案。JPMS 将 JDK 本身进行了模块化。

Jigsaw 项目

Jigsaw 项目由一个 JSR（Java Specification Request）和多个 JEP（JDK Enhancement Proposal）组成。构成 Jigsaw 的规范包括：

- JSR 376：Java 平台模块系统
- JEP 200：模块化 JDK
- JEP 201：模块化源码
- JEP 220：模块化运行时镜像
- JEP 260：封装大多数内部 API

- JEP 261：模块系统
- JEP 282：jlink：Java 连接器

Jigsaw 的项目主页（*http://openjdk.java.net/projects/jigsaw/*）包含了各个规范的链接。

Java 模块

Java 模块是在默认包中包含一个 module-info.java 文件的 JAR。因为“module-info”并不是合法的 Java 类名，所以 Java 8 及以前的版本会将其忽略。它确实会被编译，并且能够通过反射访问。模块文件声明了模块的名称、模块的依赖以及该模块导出了哪些包。该模块提供的服务也可以在该文件中声明。

模块系统具有以下的规则：

- 模块指明要导出哪些包。在这些包中的公开类型能够被其他模块访问；
- 在运行时，不能访问没有导出的包，也不能使用 Java 反射访问类型；
- 模块名必须是全局唯一的，应该使用反向的域名作为模块名称；
- 同一模块只能加载一个版本。多个版本可以使用层（java.lang.ModuleLayer）来加载；
- 模块依赖图不能出现环状；
- 所有模块的依赖必须要在启动的时候提供，任何缺失的依赖都会导致错误出现；
- 模块路径，类似于类路径，已经添加到 Java 工具中了。

217 要在 Java 9 中运行，Java 应用不一定非要进行模块化。如果代码使用类路径加载而不是模块路径，它依然会按照 Java 9 之前的方式来运行。如果代码使用模块路径加载，那么会解析模块图并且检查依赖。

对于一个 JAR 文件，模块系统如何解释它要依赖于它是在模块

路径上进行加载还是在类路径上进行加载，另外还要看它是否包含 module-info.java 文件。JPMS 将会创建应用模块、未命名模块（unnamed module）或自动模块（automatic module）。表 21-1 列出了各种行为。

表21-1　JPMS加载行为

	--module-path	-classpath
模块化 JAR	应用模块	未命名模块
非模块化 JAR	自动模块	未命名模块

自动模块

对于缺少 module-info.java 文件的 JAR 包，JPMS 会自动创建一个模块。在自动模块中，所有的包都是导出的。自动模块的名称是根据 JAR 文件的名称衍生出来的，模块名称的规则如下：

- 移除掉“.jar”后缀；
- 模块的名称将会提取分隔符前第一个满足 *-(\\d(\\.|$))+* 正则表达式的文本，在分隔符后面如果能提取版本信息，将会作为模块的版本；
- 所有非字母和数字的字符都会被点号取代，重复出现的点号会替换为单个点号，在开头和结尾的点号会被移除。

表21-2　展现了模块名称的一些样例

218

JAR 名称	模块名	版本
`forex-calc.jar`	`forex.calc`	`None`
`forex-calc-0.1.0.jar`	`forex.calc`	`0.1.0`
`forex-0.1.0.jar`	`forex`	`0.1.0`

未命名模块

从类路径下加载到的类，与模块路径不同，是以未命名模块的形式加载的。未命名模块中的类对模块路径下的类是不可见的。

可访问性

模块为 Java 添加了一个新的封装层。借助模块，我们可以限制对公开类型的访问，并且能够选择性地声明哪些模块可以访问某个包。公开的类不再是全局公开的了，public 类可以限制只能在模块中访问，或者将其导出供其他模块访问。

编译模块

编译器命令 javac 进行了扩展，添加了额外的参数来处理模块。多个模块可以同时编译。对于多个模块来说，每个模块的内容应该放到同名的目录下。样例如图 21-1 所示：

219

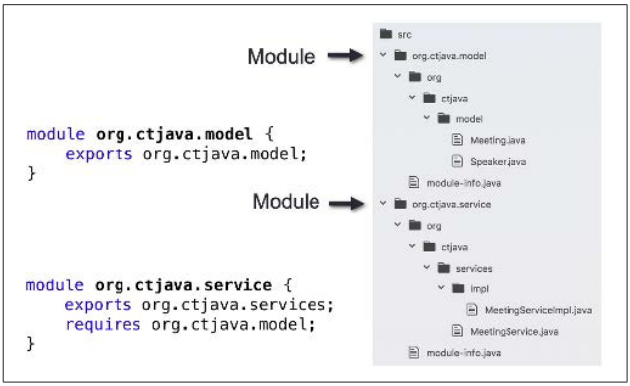


图21-1 多模块布局的样例

图 21-1 的样例可以在 UNIX 系统下使用如下命令进行编译：

```
javac -d out --module-source-path src $(find .
-name "*.java")
```

其他的模块命令行参数如下所示：

`--add-modules <module>(,<module>)*`

要解析的根模块和初始模块，如果 <module> 为 ALL-MODULE-PATH 将会解析模块路径下的所有模块。

`--limit-modules <module>(,<module>)*`
限制可观测到的模块的范围。

`--module <module-name>, -m <module-name>`
只编译特定的模块，检查时间戳。

`--module-path <path>, -p <path>`
指定去何处查找应用模块。

`--module-source-path <module-source-path>`
为多个模块指定去何处查找输入的源文件。

`--module-version <version>`
指明要编译的模块版本。

220

`--upgrade-module-path <path>`
覆盖可升级的模块的位置。

模块化 JDK

作为 Jigsaw 项目的一部分（JEP 200），JDK 本身也进行了大量的重构和模块化。java.base 模块默认会加载，这就意味着我们不需要显式地依赖它。但是，如果你使用 JavaFX、JDBC，就需要包含相关的模块，另外还要在代码中添加相关的导入。表 21-3 给出了模块的列表。

Java 编译器使用 java.se.ee 模块，而在运行时 Java 使用 java.se 模块。包含在 java.se.ee 模块中的功能一般是由 Java EE 容器提供的。

表21-3　模块概述

模块	所需的依赖
java.activation	java.base、java.datatransfer、java.logging
java.base	
java.compiler	java.base

续表

模块	所需的依赖
java.cobra	java.base、java.desktop、java.logging、java.naming、java.rmi、java.transaction、jdk.unsupported + 4 个传递性依赖
java.datatransfer	java.base
java.desktop	java.base、java.datatransfer、java.prefs、java.xml
java.instrument	java.base
java.logging	java.base
java.management	java.base
java.management.rmi	java.base、java.management、java.naming、java.rmi + 2 个传递性依赖
java.naming	java.base、java.security.sasl + 1 个传递性依赖
java.prefs	java.base、java.xml
java.rmi	java.base、java.logging
java.scripting	java.base
java.se	java.base、java.compiler、java.datatransfer、java.desktop、java.instrument、java.logging、java.management、java.management.rmi、java.naming、java.prefs、java.rmi、java.scripting、java.security.jgss、java.security.sasl、java.sql、java.sql.rowset、java.xml、java.xml.crypto

221

续表

模块	所需的依赖
java.se.ee	java.activation、java.base、java.corba、java.se、java.transaction、java.xml.bind、java.xml.ws、java.xml.ws.annotation + 19 个传递性依赖
java.security.jgss	java.base、java.naming + 2 个传递性依赖
java.security.sasl	java.base、java.logging
java.smartcardio	java.base
java.sql	java.base、java.logging、java.xml
java.sql.rowset	java.base、java.logging、java.naming、java.sql + 2 个传递性依赖
java.transaction	java.base,java.rmi + 1 个传递性依赖
java.xml	java.base
java.xml.bind	java.activation、java.base、java.compiler、java.desktop、java.logging、java.xml、jdk.unsupported + 2 个传递性依赖
java.xml.crypto	java.base、java.logging、java.xml

续表

模块	所需的依赖
java.xml.ws	java.activation、java.base、java.desktop、java.logging、java.management、java.xml、java.xml.bind、java.xml.ws.annotation、jdk.httpserver、jdk.unsupported + 3 个传递性依赖
java.xml.ws.annotation	java.base

jdeps

为了准备 Java 9 中的 Jigsaw 项目，Oracle 在 Java 8 中添加了 jdeps 命令行工具。这是一个静态的依赖检查器，它的目的是为了辅助 Java 9 的准备工作。这个工具有 3 个主要的用途：

- 识别一组类所需的 JDK 模块依赖；
- 跟踪一组类的传递性依赖；
- 识别对未文档化内部 JDK 类的依赖

这个工具可以将分析结果展现控制台上，也可以将其导出为“.dot”文件。像 graphviz（https://graphviz.org/）这样的工具可以将“.dot”文件渲染为图形化的输出。

识别依赖

```
jdeps postgresql-42.1.1.jar
    postgresql-42.1.1.jar -> /Library/Java/
    JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/
    jre/lib/jce.jar
    postgresql-42.1.1.jar -> not found
    postgresql-42.1.1.jar -> /Library/Java/
    JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/
    jre/lib/rt.jar
     org.postgresql (postgresql-42.1.1.jar)
     -> java.io
```

```
 -> java.lang                                                    223
 -> java.net
 -> java.security
 -> java.sql
 -> java.util
 -> java.util.logging
 -> org.postgresql.copy postgresql-42.1.1.jar
 -> org.postgresql.fastpath postgresql-42.1.1.jar
 -> org.postgresql.jdbc postgresql-42.1.1.jar
 -> org.postgresql.largeobject postgresql-42.1.1.jar
 -> org.postgresql.replication  postgresql-42.1.1.jar
 -> org.postgresql.util postgresql-42.1.1.jar
...
```

识别未文档化的 JDK 内部依赖

要识别对未文档化（undocumented）的 JDK 类的依赖，可以使用 –jdkinternals。未文档化的 JDK 类指的是以“com.sun.* ”或“sun.*”开头的类。这些类不应该在 JDK 之外使用，可能随时被移除。在 Java 9 中，很多这样的 API 已经重构或移除了。

如下的 jdeps 命令返回了 MyEncoder jar 的依赖：

```
jdeps -jdkinternals MyEncoder.jar
MyEncoder.jar -> /Library/Java/JavaVirtualMachines/
jdk1.8.0_131.jdk/Contents/Home/jre/lib/rt.jar
 org.ctjava.util.TransmitUtil (MyEncoder.jar)
 -> sun.misc.BASE64Encoder JDK internal API (rt.jar)

Warning: JDK internal APIs are unsupported and private
to JDK implementation that are
subject to be removed or changed incompatibly and
could break your application.
Please modify your code to eliminate dependency on any   224
JDK internal APIs.
For the most recent update on JDK internal API
replacements, please check:
https://wiki.openjdk.java.net/display/JDK8/
Java+Dependency+Analysis+Tool

JDK Internal API Suggested Replacement
---------------- ---------------------
sun.misc.BASE64Encoder Use java.util.Base64 @since 1.8
```

在本例中，TransmitUtil 类依赖了未文档化的 JDK 类，而这个类在 Java 9 中已经不支持了。

定义模块

要定义一个模块，必须在默认的包中创建 module-info.java 文件，该文件的内容格式如下所示：

```
<open> module <module-name> {
[export <java package> [to <module name>]
 [requires [transitive] <module-name>]
 [opens <module name> [to <module name]]
[provides <interface> with <implementation>]
[uses <interface>]
}
```

模块的名称必须是唯一的，应该使用反向域名的模式。如下的样例定义了名为 org.ctjava.admin 的模块，它没有导出任何包也没有依赖任何包：

```
module org.ctjava.admin {
}
```

导出包

包中所有的公开类型都可以导出，我们只需添加如下的导出语句到模块定义中：

225

```
module org.ctjava.admin {
  exports org.ctjava.admin.api
}
```

在本例中，org.ctjava.admin.api 包下所有的公开类都进行了导出，依赖 org.ctjava.admin 的其他模块都可以访问到它们。

包可以有选择性地导出给其他的模块，比如：

```
module org.ctjava.admin {
  exports org.ctjava.admin.ui to javafx.graphics;
}
```

在本例中，org.ctjava.admin.ui 有选择性地导出给了 javafx.graphics。这样，javafx.graphics 中的类在访问 org.ctjava.admin.ui 中的类时，就不需要在模块文件中声明对这个包的依赖了。当另外的包使用反射时，这种用法非常常见。如果不进行导出，

javafx.graphics 中的类将不能对 org.ctjava.admin.ui 中的类执行反射。

声明依赖

要声明对其他包的依赖，需要向模块定义中添加 requires 语句。如下的样例有 3 个依赖，当模块加载的时候，这些依赖必须都要存在：

```
module org.ctjava.admin {
    requires javafx.controls;
    requires org.ctjava.services;
    requires org.ctjava.message.api;
}
```

传递性依赖

如果某个模块的导出包中使用了其他模块的类，并且这个导出包要被下游的其他模块所使用，那么这个模块必须要在依赖中添加 transitive 关键字。 226

考虑如下的模块定义：

```
module org.ctjava.services {
    requires transitive org.ctjava.model;
    exports org.ctjava.services;
}
```

这个定义导出了 org.ctjava.services 包，这个包中包括如下的类：

```
public interface MeetingService {
    void scheduleMeeting(Meeting meeting,
            Date scheduledDate);
    void updateMeeting(Meeting meeting);
    …
    }
}
```

这个类用到了来自 org.ctjava.model 包的 Meeting 类型。如果 org.ctjava.services 在对 org.ctjava.model 的依赖中，不在 requires 上添加 transitive 关键字，那么依赖 org.ctjava.services

的用户将无法访问 Meeting 类，因此也就无法调用或使用 MeetingService 类。

定义服务提供者

模块可以导出服务，这些服务可以在启动的时候动态添加到模块路径中。服务提供者 API（Service Provider API）最初是在 Java 6 中加入的，并针对 Java 9 进行了修改。我们按照如下的方式来使用服务提供者功能：

- 某个模块包含服务的接口定义；
- 一个或多个模块包含服务的实现；
- 某个模块使用该服务。

227 图 21-1 展现了样例服务的实现。模块如下所示：

org.ctjava.message.api 包含了接口 MessageService.java；org.ctjava.email 包含了 MessageService.java 的实现；org.ctjava.admin 使用添加到模块路径中的 org.ctjava.message.api 实现。

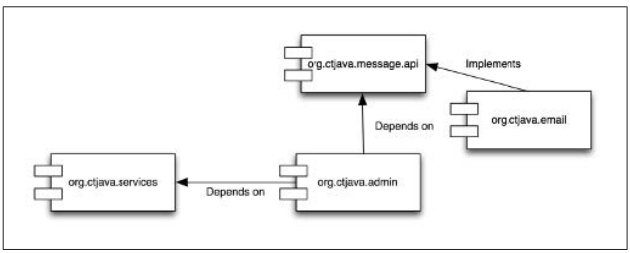


图21-2　服务提供者API样例

定义服务 API

模块 org.ctjava.message.api 中包含了 MessageService.java 接口，该接口定义了服务的协议：

```
module org.ctjava.message.api {
    exports org.ctjava.message.api;
```

```
}
```

包 org.ctjava.message.api 中包含了定义服务的接口：

```
package org.ctjava.message.api;
public interface MessageService {
    void sendMessage(String memeber, String mes
sage);
}
```

实现服务 API

228

提供服务的实现非常简单。在模块文件中，要声明对服务 API 的依赖并指明它提供了该接口的一个实现。这里需要新的 requires 和 provides 关键字来声明依赖，如下所示：

```
module org.ctjava.email {
    requires org.ctjava.message.api;
    provides org.ctjava.message.api.MessageService
            with org.ctjava.email.EmailMessageService;
}
```

服务的实现非常简单：

```
package org.ctjava.email;

import org.ctjava.message.api.MessageService;

public class EmailMessageService implements
MessageService {
    @Override
    public void sendMessage(String memeber, String
message) {
        // send message
    }
}
```

使用服务提供者

要使用某个服务的时候，需要声明对 API 的依赖，然后使用 uses 语句指明服务接口，如下所示：

```
module org.ctjava.admin {
    requires org.ctjava.message.api;
    uses org.ctjava.message.api.MessageService;
}
```

229 在使用服务的时候，借助 java.util.ServiceLoader 获取引用：

```
Iterable<MessageService> mservice = ServiceLoader.
load(MessageService.class);
for(MessageService ms : mservice) {
  ms.sendMessage(member, “Hello World!” );
}
```

jlink

jlink 工具能够将一组模块及其依赖组装并优化到一个自定义的运行时镜像中。这个工具是在 JEP 282 中定义的，只有所需的模块才会包含到镜像中。例如，使用 JavaFX 的桌面应用只有 95 MB，而与之相对的是整个 JDK 有 454 MB。

jlink 命令具有如下的参数：

```
jlink --module-path <modulepath> +
        --add-modules <modules> +
        --limit-modules <modules> +
        --output <path>
```

下面是一个样例：

```
jlink --module-path $JAVA_HOME/jmods:dist --add-
modules org.ctjava.TimerUtil --output test
```

第 3 部分

附录

附录 A

Fluent API

233

Fluent API，也被称为 Fluent 接口，它是一种面向对象的 API，其设计目的是让 API 代码更加易读，因此也会更加易用。将对象以链的方式装配在一起有助于实现可读性和可用性的目标。按照这种设计，在如下的样例中，链式的方法维持了相同的类型。

```
/*
 * 将 “‘palindrome!” 字符串转换为 “semordnilap”

// StringBuilder API
StringBuilder sb = new StringBuilder( “palindrome!” );

// 链式方法（删除、附件和反转）
sb.delete(10, 11).append( “s” ).reverse();

System.out.println( “New value: “ + sb);
$ New value: semordnilap
```

在 Java 中，有一些流行的 Fluent API，比如 Java 面向对象的查询（Java Object Oriented Querying，jOOQ）API、jMock 测试 API、Calculon Android 测试 API、Apache Camel 集成模式 API、Java 8 的 Date Time API（JSR 310）以及 Java 的 Money and Currency API（JSR 354）。上述的 API 都可以视为包含了一个 Java 领域特定语言（domain-specific language，DSL）。

234

通过使用 Fluent API，外部的 DSL 可以很容易地匹配到新的

Java 内部 DSL。

Fluent API 常用的方法前缀包括 `at`、`format`、`from`、`get`、`to` 和 `with`。

在这里以 Date Time API 中的 `LocalDateTime` 为例，介绍了链式方法和非链式方法的操作：

```
// 具有方法链的静态方法
LocalDateTime ldt2 = LocalDateTime.now()
  .withDayOfMonth(1).withYear(1878)
  .plusWeeks(2).minus(3, ChronoUnit.HOURS);
System.out.println(ldt2);

$ 1878-02-15T06:33:25.724

// 没有方法链的独立静态方法
LocalDateTime ldt1 = LocalDateTime.now();
System.out.println(ldt1);

$ 2016-11-06T16:10:12.344
```

提示

读者如果希望了解 DSL 的完整信息，可以参阅 Martin Fowler 的《领域特定语言》（Addison-Wesley）一书。

附录 B

第三方工具 235

现在有各种开源和商业的第三方工具与技术帮助我们开发基于 Java 的应用。

这里所列出的样本集都是非常有效和流行的，起码经常会用到。如果要在商业限制的环境下使用，别忘了检查开源工具的许可证协议。

开发、CM（配置管理）和测试工具

Ant（*http://ant.apache.org/*）

Apache Ant 是一个基于 XML 的工具，用来构建和部署 Java 应用。它非常类似于众所周知的 Unix make 工具。

Bloodhound（*http://bloodhound.apache.org/*）

Apache Bloodhound 是一个开源的基于 Web 的项目管理和 bug 跟踪系统。

CruiseControl（*http://cruisecontrol.sourceforge.net/*）

CruiseControl 是一个用于持续构建过程的框架。

Enterprise Architect（*http://www.sparxsystems.com/*） 236

Enterprise Architect 是一个商用的计算机辅助软件工程

（computer-aided software engineering，CASE）工具，它提供了使用 UML 进行 Java 代码正向和反向工程的功能。

FindBugs（*http://findbugs.sourceforge.net/*）

FindBugs 是一个在 Java 代码中查找 bug 的程序。

Git（*https://git-scm.com/*）

Git 是一个开源的分布式版本控制系统。

Gradle（*https://gradle.org/*）

Gradle 是一个构建系统，提供了对测试、发布和部署的支持。

Hudson（*http://hudson-ci.org/*）

Hudson 是一个可扩展的持续集成服务器。

Ivy（*https://ant.apache.org/ivy/*）

Apache Ivy 是一个传递性依赖的管理器，它能够与 Apache Ant 集成在一起。

Javacc（*https://javacc.org/*）

Javacc 工具能够读取语法规范并将其转换为 Java 应用，该应用能够识别代码的匹配情况。

Jalopy（*http://jalopy.sourceforge.net/*）

Jalopy 是一个针对 Java 的源码格式化器，它具有针对 Eclipse、jEdit、NetBeans 和其他工具的插件。

jClarity（*http://www.jclarity.com/*）

jClarity 是一个适用于云环境的性能分析和监控工具。

jEdit（*http://www.jedit.org/*）

jEdit 是一个面向程序员的文本编辑器，它通过一个插件管理器支持多种插件。

JavaFX Scene Builder（*http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html*）

JavaFX Scene Builder 是设计 JavaFX 应用的可视化布局工具。

Jenkins（*https://jenkins.io/*）

Jenkins CI 是一个开源的持续集成服务器，之前被称为 Hudson Labs。

JIRA（*https://www.atlassian.com/software/jira*）

JIRA 是一个商用的 bug 跟踪、issue 跟踪以及项目管理的应用。

JUnit（*https://junit.org/*）

JUnit 是一个用于单元测试的框架，它提供了编写和运行可重复测试的一种方式。

JMeter（*http://jmeter.apache.org/*）

Apache JMeter 是一个测量系统行为的应用，这些行为包括功能和性能。

Maven（*http://maven.apache.org/*）

Apache Maven 是一个软件项目管理的工具，Maven 能够管理构建、报告和文档。

Nemo（*https://about.sonarcloud.io/*）

Nemo 是一个在线的 Sonar 实例，专门用于开源的项目。

PMD（*http://pmd.sourceforge.net/*）

PMD 能够扫描 Java 源码，查找 bug、欠佳的代码和过于复杂的表达式。

SonarQube（*https://www.sonarqube.org/*）

SonarQube 是一个开源的质量管理平台。

Subversion（*http://subversion.apache.org/*）

Apache Subversion 是一个中心化的版本控制系统，能够跟踪针对一组文件的工作和变更。

库

ActiveMQ（*http://activemq.apache.org/*）

Apache ActiveMQ 是一个消息代理，支持多种跨语言的客户端和协议。

238 *BIRT*（*https://www.eclipse.org/birt/*）

BIRT 是一个开源的基于 Eclipse 的报表系统，可以用到 Java EE 应用中。

Bitlyj（*https://github.com/criedel/bitlyj*）

支持 Bitly 短 URL 服务的 DSL。

Camel（*http://camel.apache.org/*）

Apache Camel 是一个基于规则的路由和中介（mediation）引擎。

gedcom4j（*http://gedcom4j.org/main/*）

gedcom4j 是用来解析、操作和写入 GEDCOM 数据的 Java 库。

Geocoder-java（*https://github.com/panchmp/geocoder-java*）

Geocoder-java 是针对 Google geocoder v3 的 Java API。

GSON（*https://github.com/google/gson*）

Google-gson 是能够在 Java 对象和 JSON 之间互相转换的 Java API。

Guava（*https://github.com/google/guava*）

Google Guava 是一组库，它包含了新的集合类型、不可变集合、一个图的库、函数式类型、基于内存的缓存、并发工具、I/O、哈希、原始类型与反射。

Hibernate（*http://hibernate.org/*）

Hibernate 是一个对象 / 关系持久化和查询服务。它允许开发持久化的类。

iText（*https://itextpdf.com/*）

iText 是一个允许创建和操作 PDF 文档的 Java 库。

Jakarta Commons（*http://commons.apache.org/*）

Jakarta Commons 是一个可重用 Java 组件的仓库。

Jackrabbit（*http://jackrabbit.apache.org/jcr/index.html*）

Apache Jackrabbit 是一个内容仓库系统，它提供了层级化的内容存储和控制功能。

JasperReports（*https://community.jaspersoft.com/project/jasperreports-library*）239

JasperReports 是一个开源的 Java 报表引擎。

Jasypt（*http://www.jasypt.org/*）

Jasypt 是一个 Java 库，允许开发人员添加基本的加密功能。

JFreeChart（*http://www.jfree.org/jfreechart/*）

JFreeChart 是一个能够生成图表的 Java 类库。

JFXtras2（*http://jfxtras.org/*）

JFXtras2 是一组面向 JavaFX 2.0 的控件和插件集。

JGoodies（*http://www.jgoodies.com/*）

JGoodies 为常见的用户界面任务提供了组件和解决方案。

JIDE（*http://www.jidesoft.com/*）

JIDE 软件提供了各种 Java 和 Swing 组件。

jMonkeyEngine（*http://jmonkeyengine.org/*）

jMonkeyEngine 是一个库的集合，提供了 Java 3D（OpenGL）游戏引擎。

JOGL（*https://jogamp.org/jogl/www/*）

JOGL 是一个支持 OpenGL 和 ES 规范的 Java API。

jOOQ（*http://www.jooq.org/*）

jOOQ 是一个 Fluent API，用来实现类型安全的 SQL 构建和执行。

Moneta（*https://javamoney.github.io/ri.html*）

Moneta 是 JSR 354 Money & Currency API 的一个参考实现。

opencsv（*http://opencsv.sourceforge.net/*）

opencsv 是面向 Java 的逗号分隔值（comma-separated value，CSV）的解析库。

240 *POI*（*http://poi.apache.org/*）

Apache Poor Obfuscation Implementation（POI）是一个读取和写入 Microsoft Office 格式的库。

ROME（*https://rometools.github.io/rome/*）

ROME 是一个用于 RSS 和 Atom feed 的 Java 框架。

RXTX（*http://users.frii.com/jarvi/rxtx/*）

RXTX 为 Java 提供了原生的串行和并行的通信。

Spring（*https://spring.io/*）

Spring 是分层的 Java/Java EE 应用框架。

Tess4J（*http://tess4j.sourceforge.net/*）

针对 Tesseract 光学字符识别（optical character recognition，OCR）API 的 Java JNA 包装器。

Twitter4j（*http://twitter4j.org*）

针对 Twitter API 的 Java 库。

集成开发环境

BlueJ（*https://www.bluej.org/*）

BlueJ 是针对入门教学所设计的 IDE。

Eclipse IDE（*https://www.eclipse.org/*）

Eclipse IDE 是一个用于创建桌面、移动和 Web 应用的开源 IDE。

Greenfoot（*https://www.greenfoot.org/door*）

Greenfoot 是一个简单的 IDE，用于 Java 面向对象编程的教学。

IntelliJ IDEA（*https://www.jetbrains.com/idea/*）

IntelliJ IDEA 是一个用于创建桌面、移动和 Web 应用的商业 IDE。

JCreator（*http://www.jcreator.com/*）

JCreator 是一个用于创建桌面、移动和 Web 应用的商业 IDE。

JDeveloper（*http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html*）

JDeveloper 是 Oracle 的 IDE，能够用于创建桌面、移动和 Web 应用。 241

NetBeans IDE（*https://netbeans.org/*）

NetBeans 是 Oracle 的开源 IDE，能够用于创建桌面、移动和 Web 应用。这个 IDE 目前处于 Apache Incubator。

Web 应用平台

Geronimo（*http://geronimo.apache.org/*）

Apache Geronimo 是一个 Java EE 服务器，用于应用、门户和 Web 服务。

Glassfish（*https://javaee.github.io/glassfish/*）

Glassfish 是一个开源的 Java EE 服务器，用于应用、门户和 Web 服务。Payara 是 Glassfish 的一个变种。

IBM WebSphere（*https://www.ibm.com/cloud/websphere-application-platform*）

IBM WebSphere 是商用的 Java EE 服务器，用于应用、门户和 Web 服务。

JavaServer Faces（*https://jcp.org/en/jsr/detail?id=314*）

JavaServer Faces 简化了 Java 服务器应用的用户界面构建。JSF 的实现及组件集包括 Apache MyFaces、ICEFaces、RichFaces 和 Primefaces。

Jetty（*https://www.eclipse.org/jetty/*）

Jetty 是一个用于 Java Servlets 和 JavaServer Pages 的 Web 容器。

Oracle WebLogic Application Server（*https://www.oracle.com/middleware/weblogic/index.html*）

Oracle WebLogic Application Server 是一个商用的 Java EE 服务器，用于应用、门户和 Web 服务。

Resin（*http://www.caucho.com/*）

Resin 是一个高性能、针对云进行了优化的 Java 应用服务器。

242 *ServiceMix*（*http://servicemix.apache.org/*）

Apache ServiceMix 是一个企业级的服务总线，它在 Java 业务集成规范（Java Business Integration）之上结合了面向服务架构和事件驱动架构的功能。

Sling（*http://sling.apache.org/*）

Sling 是一个 Web 应用框架，它采用了表述性状态转移（Representational State Transfer，REST）的软件架构风格。

Struts（*http://struts.apache.org/*）

Apache Struts 是一个创建企业级 Java Web 应用的框架，它采用了模型 - 视图 - 控制器的架构。

Tapestry（*http://tapestry.apache.org/*）

Apache Tapestry 是基于 Java Servlet API 创建 Web 应用的框架。

Tomcat（*http://tomcat.apache.org/*）

Tomcat 是一个用于 Java Servlets 和 JavaServer Pages 的 Web 容器。

TomEE（*http://tomee.apache.org/*）

TomEE 是一个完整的 Apache Java EE 6 Web Profile 认证的栈。

WildFly（*http://www.wildfly.org/*）

WildFly 之前被称为 JBoss Application Server，是一个开源的 Java EE 服务器，用于应用、门户和 Web 服务。

兼容 JSR-223 的脚本语言

BeanShell（*http://www.beanshell.org/*）

BeanShell 是一个嵌入式的 Java 源码解释器，具有基于对象的脚本语言特性。

Clojure（*https://www.clojure.org/*）

243

Clojure 是一个面向 Java 虚拟机（Java Virtual Machine）、通用语言运行时（Common Language Runtime）和 JavaScript 引擎的动态编程语言。

FreeMarker（*https://freemarker.apache.org/*）

FreeMarker 是基于 Java 的通用模板引擎。

Groovy（*http://www.groovy-lang.org/*）

Groovy 是一个脚本语言，它在类似 Java 风格的语法中添加了很多 Python、Ruby 和 Smalltalk 的特性。

Jacl（*http://tcljava.sourceforge.net/docs/website/index.html*）

Jacl 是 Tcl 脚本语言的纯 Java 实现。

JEP（*http://www.singularsys.com/jep/*）

Java Math Expression Parser（JEP）是解析和执行数学表达式的 Java 库。

Jawk（*http://jawk.sourceforge.net/*）

Jawk 是 AWK 脚本语言的纯 Java 实现。

Jelly（*http://commons.apache.org/proper/commons-jelly/*）

Jelly 是用来将 XML 转换为可执行代码的 Java 实现。

JRuby（*http://jruby.org/*）

JRuby 是 Ruby 编程语言的纯 Java 实现。

Jython（*http://www.jython.org/*）

Jython 是 Python 编程语言的纯 Java 实现。

Nashorn（*http://openjdk.java.net/projects/nashorn/*）

Nashorn 是 JavaScript 语言的实现。它是唯一一个在 Java 脚本 API（Java Scripting API）中默认包含脚本引擎实现的脚本语言。

244 *Scala*（*http://www.scala-lang.org/*）

Scala 是一个通用的编程语言，其设计目标是以一种简洁、优雅和类型安全的方式描述通用的语言模式。

Sleep（*http://sleep.dashnine.org/*）

Sleep 基于 Perl，是一门针对 Java 应用的嵌入式脚本语言。

Velocity（*http://velocity.apache.org/*）

Apache Velocity 是基于 Java 的通用模板引擎。

Visage（*https://code.google.com/archive/p/visage/*）

Visage 是一门领域特定语言（DSL），用来表述用户界面编写的目标。

附录 C

UML基础 245

通用建模语言（Unified Modeling Language，UML）是一门对象建模的特定语言，它使用图形标记的方式创建系统的抽象模型。对象管理组织（Object Management Group，OMG，*http://www.uml.org/*）在管理着 UML。这个建模语言能够应用到 Java 程序上，帮助图形化地阐述诸如类关系、序列图等内容。UML 的最新规范可以在 OMG 的 Web 站点（*http://www.omg.org/spec/UML*）上查阅。关于 UML 可以参考一本很有助益的图书，即 Martin Fowler 所编写的 *UML Distilled*，第 3 版（Addison-Wesley）。

类图

类图阐述了系统的静态结构，展现了类以及它们之间的关联关系。一个类图分为 3 个区间，分别是名称、属性（可选）和操作（可选）。参见图 C-1 及其实例。

246

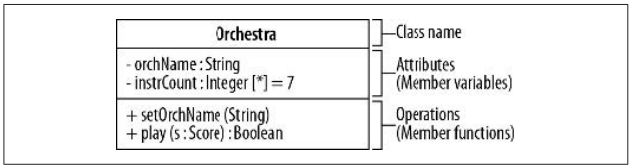


图 C-1　类图

```
// 对应的代码片段
class Orchestra { // 类名
  // 属性
  private String orch Name;
  private Integer instrCount = 7;
  // 操作
  public void setOrchName(String name) {...}
  public Boolean play(Score s) {...}
}
```

名称

名称区间是必需的，以加粗的格式显示类或接口的名称。

属性

属性区间是可选的，包含了代表这个对象状态的成员变量。完整的 UML 用法如下所示：

```
visibility name : type [multiplicity] = defaultValue
{property-string}
```

通常来讲，只会展现属性名和类型。

操作

操作区间是可选的，包含了代表这个对象行为的成员函数。完整的 UML 用法如下所示：

247

```
visibility name (parameter-list) :
return-type-expression
{property-string}
```

通常来讲，只会展现操作的名称和参数列表。

提示

{property-string} 可以是一些可选的属性，比如 {ordered} 或 {read-only}。

可见性

可见性指示器（前缀符号）是可选的，它用来定义访问修饰符。这个指示器可以用到类图的成员变量和成员函数上，参见表 C-1。

表C-1

可见性指示器	访问修饰符
~	包私有
#	protected
-	private

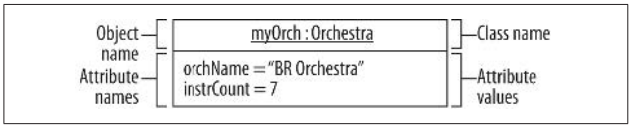
对象图

对象图与类图的区别之处在于对象名称部分要使用下画线。文本可以按照 3 种方式来表述，参见表 C-2 ：

表C-2　对象名

: ClassName	只有类名
objectName	只有对象名
objectName : ClassName	对象名和类名

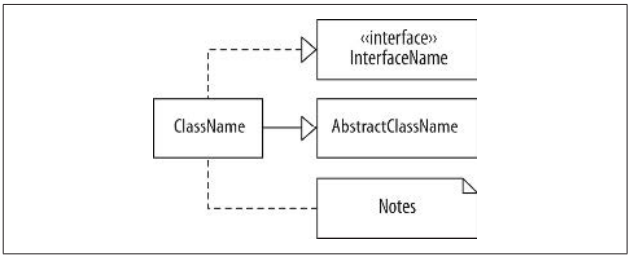
对象图并不常用，但是它可以用来展现详细信息，如图 C-2 所示：248



图C-2　对象图

图形化的展现方式

在 UML 图中，主要的构建块就是图形化的图标，参见图 C-3 ：



图C-3　图形化图标表述

类、抽象类和接口

类、抽象类和接口都在矩形框内以加粗字体显示。抽象类同时要斜体显示，接口要添加 interface 前缀，并且要放到书名号中。书名号适用于原型和接口的场景。

提示

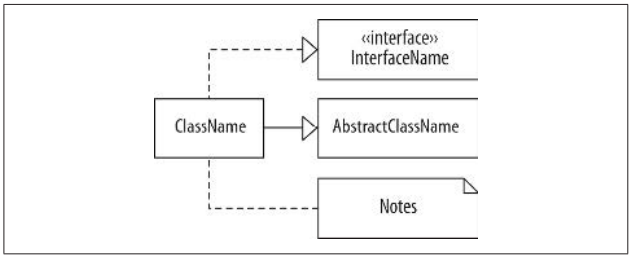
提示是以矩形显示的注释，它们的角是折叠的。提示可以单独展示，也可以通过虚线连接到另外一个图标上。

249

包

包通过类似文件夹的图标来展示。包名位于较大的区间内，除非较大的区间被其他图标（如类的图标）所占据。在后者的情况中，可以将包名放到较小的区间内。带有虚线的开放箭头用来展现包的依赖关系。

箭头要指向被依赖的包，图 C-4 展示了包图：



图C-4　包图

连接器

连接器是图形化的图片，展现了类之间的关联关系。关于连接器的更多细节，参见下面“类的关联关系”一节。

多重指示器

多重指示器代表了有多少类参与到了关联之中（参见表 C-3）。这些指示器通常和连接器一起使用，还可以在属性区间中作为成员变量的一部分。

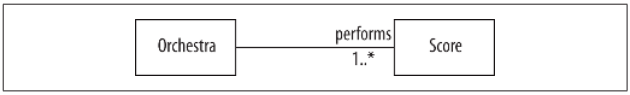
表C-3　多重指示器

250

指示器	定义
*	零个或多个对象
0..*	零个或多个对象
0..1	可选（零个或一个对象）
0..n	零到 n 个对象，其中 n>1
1	一个对象
1..*	一个或多个对象
1..n	一到 n 个对象，其中 n>1
m..n	指定对象数量的范围
n	n 个对象，其中 n>1

角色名称

如果在类之间需要进一步明确关系，会用到角色名称。角色名称通常会与多重指示器一同使用。图 C-5 展示的 Orchestra 要 performs 一个或多个 Score。

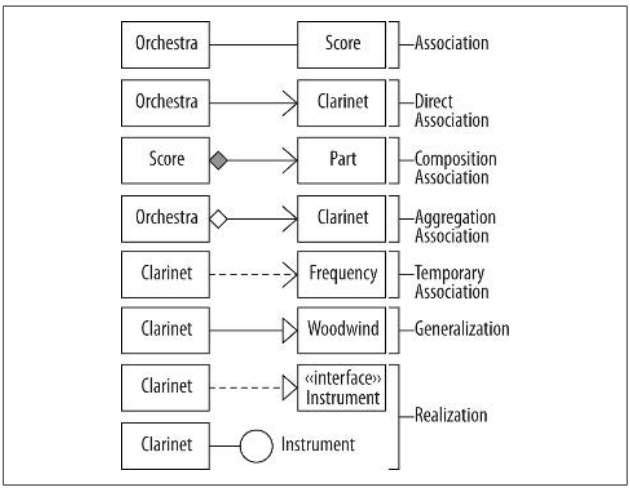


图C-5　角色名称

251

类的关联关系

类的关联关系要通过连接器和类图来表示（参见图 C-6）。在阐述关系时，图形化的图标、多重指示器以及角色名称都可能会用到。



图C-6　类的关联关系

关联

关联（association）表示了类之间的关系，可以双向描述。类属性和相关的多重性指示器可以放到目标端。

直接关联

直接关联（direct association），也被称为*导航性*（navigability），这种关联关系直接将源类连接到目标类上。我们可以将这种关联关系读作“Orchestra 有一个 Clarinet”。类属性和相关的多重指示器可以放到目标端。导航性在类之间可以是双向的。

组合关联

252

组合关联（composition association），也被称为*包含*（containment），它所建模的是整体和局部的关系，其中整体会管理局部的生命周期。各个局部不能独立存在，只能作为整体的组件。这是比聚合更强的一种关联关系。这种关系可以读作“Score 由一个或多个部分组成”。

聚合关联

聚合关联（aggregation association）建模的是整体和局部的关系，其中局部可以独立于整体而存在。整体不管理局部的存在性。这种关系可以读作“Orchestra 是整体，Clarinet 是 Orchestra 的一部分”。

时态关联

时态关联（temporary association），也被称为*依赖*（dependency），它代表了某个类需要确保另外一个类存在。这种关系常见的场景是用作本地变量、返回值或成员函数的参数。把一个 frequency 传递给 Clarinet 类的 tune 的方法时，我们可以将其读作 Clarinet 类依赖于 Frequency 类，或者“Clarinet 使用了

Frequency”。

泛化

泛化（generalization）指的是某个特定的类继承一个更为通用的类。在 Java 中，我们知道这就是继承，比如一个类扩展 Woodwind，或者说“Clarinet 是一个 Woodwind”。

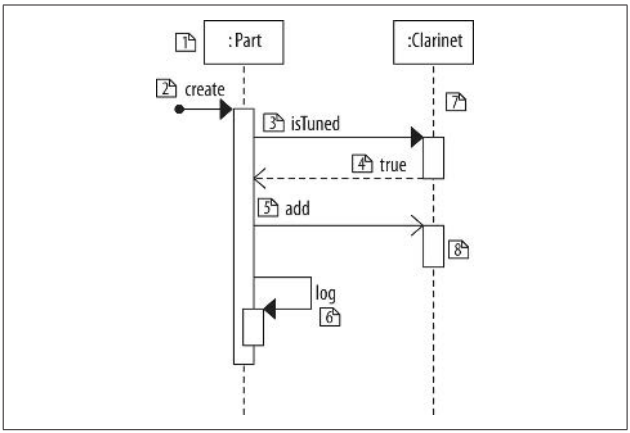
实现

实现（realization）建模的是一个类实现了某个接口，比如 Clarinet 类实现了 Instrument 接口。

序列图

UML 序列图用来展示对象之间的动态交互，参见图 C-7。这种协作关系从图的顶部开始，然后一直向下延伸到底部。

253



图C-7　序列图

参与者（1）

参与者（participant）可以想象为对象。

初始消息（2）

初始消息（found message）是调用者在图中未展示的消息。这意味着发送者是未知的，或者不需要在给定的图上展现。

同步消息（3）

如果源需要等待目标完成消息处理，就需要使用同步消息（synchronous message）。

返回调用（4）

返回调用（return call）可以有选择性地描述返回值，但是它通常会排除在序列图之外。

异步消息（5）

如果源不需要等待目标处理完消息，就可以使用异步消息（asynchronous message）。 254

发给自身的消息（6）

发给自身的消息（message to self），也被称为自我调用（self-call），它所定义的是对象内部的消息。

生命线（7）

生命线（lifeline）是与每个对象关联的，并且垂直显示。它们是与时间相关的，并且按顺序向下读取，最早的事件发生在页面的顶部。

活动条（8）

活动条（activation bar）标记在时间线或其他活动条上。活动条代表了参与者（对象）在协作的过程中何时处于活跃状态。

索引

Symbols

!= operator, 41
$ (dollar sign), 14
( ) (parenthesis), 14
. (dot) operator, 49
    accessing class methods and members in JShell, 204
/! command (JShell), 202
/-<n> command (JShell), 202
/drop command (JShell), 205
/edit commands (JShell), 206
/list command (JShell), 204
/types command (JShell), 204
; (semicolon), 63
    in JShell expressions and statements, 200
<> (angle brackets)
    << >> angle quotes, 15, 248
    bracket separators, 15
    diamond operator, 164
== operator, 41
    use by equals() method, 42
@ (annotation) symbol, 61
[ ] (square brackets), 14
_ (underscore symbol), 14
_ keyword, 14
{ } (curly brackets), 14
λEs (see lambda expressions)
–0.0 entity, 26
… (ellipsis) in vararg method signatures, 54

A

abstract classes, 55, 89
abstract methods, 56, 89
access modifiers, 88
    and their visibility, 88
    visibility indicators in UML, 247
accessor methods, 47
acronyms, naming conventions for, 3
activation bar (UML), 254
affine objects, 102
aggregation association of classes, 252
algorithms, optimizing, 156
American Standard Code for Information Interchange (see ASCII)
annotated functional interfaces, 192
annotations, 59-61
    built-in, 59

developer-defined, 60
naming conventions for, 3
anonymous inner classes, converting to lambda expressions, 191
Apache Camel API, 234
argument list, 54-55
arithmetic operators, 15
arrays, default values of, 37
ASCII, 9-11
compact strings feature, 11
nonprintable, 11
printable, 10
assertions, 70
assignment operators, 15
association of classes, 251
asynchronous message (UML), 254
autoboxing, 31
AutoClosable interface, 82
autoconversion, 64
automatic modules, 217

B

base libraries (Java), 97-99
Big O notation, 157
binary data
reading from files, 132
reading from sockets, 134
writing to files, 133
writing to sockets, 134
binary literals, 18
binary numeric promotion, 29
bitwise operators, 15
blocks, 64
applying locks to, 147
execution in JShell, 201
static initializer, 57
boolean literals, 17, 24
Boolean type, 64
bounds (generic type parameters), 166
bracket separators, 14
break statement, 66, 68
JShell and, 199
BufferedInputStream, 132, 134
BufferedOutputStream, 133, 135
BufferedReader, 131, 133
built-in annotations, 59
byte type, 25
in switch statements, 66
Byte type, 66

C

Calculon Android API, 234
calendars
ISO 8601 calendar system, 181-186
java.util.Calendar, 180
regional, 180
Canvas classes, 101
case statement, 66
catch block, 74
in try-catch statements, 79
multi-catch clause, 82
Certificate Revocation Lists (CRLs), 104
char type, 24
in switch statements, 66
Character class, 10, 66
character data
reading from files, 131
reading from sockets, 133
writing to files, 132
writing to sockets, 134
character literals, 17
Character.isJavaIdentifierStart(int), 14
checked exceptions, 74
common types of, 75
in JShell, 209
programmer-defined, 83
ChronoLocalDate interface, 180
Chronology interface, 180
class diagrams (UML), 245-247
attributes compartment, 246

name compartment, 246
operations compartment, 246
visibility indicators, 247
ClassCastException, 39
classes, 47-54
abstract, 55, 89
accessing methods/data members of, 49
constructors, 51
containment of, 251
data members and methods, 48
declaring in JShell, 203
dependency of, 252
generic, 163
generic methods in, 169
hierarchy and scope in JShell, 211
I/O class hierarchy, 130
implementing interfaces, 58
instantiating, 48
naming conventions for, 4
operators, 15
overloading methods, 49
overriding methods, 50
private data, accessing, 47
relationships between, in UML, 250-252
representing in UML, 248
superclasses and subclasses, 51-53
syntax, 48
this keyword, 53
classpath argument, 118
CLASSPATH environment variable, 118
Clock class, 183
clone() method, 44
cloning objects, 44
shallow and deep cloning, 45
closures (see lambda expressions)
CM, third-party tools for, 235-237
code snippets (see JShell; snippets)
Collection.parallelStream(), 155
Collection.stream(), 155
collections
concurrent, 150
defined, 153
Collections Framework, 5, 153-160
collection algorithm inefficiencies, 156
Collection interface and subinterfaces, 153
valuable methods, 155
collection type implementations, 154
Collections class algorithms, 155
Comparator functional interface, 158-161
convenience factory methods, 161
generics, 163
command-line tools, 112
compiler, 112-114
executing JAR files, 117
for garbage collection, 122-125
for memory management, 122-125
JAR, 116
Java interpreter, 114
-X options, 114
commands (JShell), summary of, 212
comments, 12
Common Object Request Broker Architecture (CORBA), 103
compact strings, 11
Comparator functional interface, 158-160, 191
comparison operators, 15
compiler (javac), 112-114

compiling modules, 218
composition association of classes, 251
compressed files, 136
concurrency, 143-151
collections, 150
creating threads, 143-145
executor utilities, 148
methods for, 145
synchronized statements and, 147
synchronizers, 150
thread priorities, 145
thread states, 145
timing utility, 151
concurrent mark-sweep (CMS) collector, 121
conditional operators and numeric promotion of primitive types, 30
conditional statements, 64
if else if statement, 65
if else statement, 65
if statement, 64
switch statement, 66
connectors (UML), 249
Console class, 130
constants
naming conventions for, 4
static, 57
constructors, 51
calling from another constructor in same class using this, 53
calling superclass constructor with super keyword, 52
for user-defined exceptions, 84
lambda expressions and, 191
with generics, 165
containment of classes, 251
continue statement, 69
JShell and, 199
conversion of reference types, 38
narrowing conversions, 39
widening conversions, 39
copy constructors, 45
CORBA libraries (Java), 103
CRLs (see Certificate Revocation Lists)
currency symbols, 21

D

data members
accessing in objects, 49
final, 89
in classes, 48
static, 56, 90
transient, 90, 135
data structures, optimizing, 156
DataInputStream, 132, 134
DataOutputStream, 133, 134
Date and Time API (JSR 310), 179-186, 234
durations and periods, 184
formatting date-time objects, 186
ISO calendar, 181
JDBC and XSD mapping, 185
legacy code and, 180
LocalDateTime class, with and without method chaining, 234
machine interface for, 183
primary classes, 181
regional calendars in, 180
DateTimeFormatter class, 186
Debian, 109
decimal integers, 18
deep cloning, 45
default method, 89
default statement, 66
default values (of reference types), 36-38
arrays, 37

instance and local variables, 36
defender method, 89
dependencies, 252
declaring, 225
in JShell, 200
transitive, 226
dependency checker (jdeps), 222-224
identifying dependencies, 222
identifying undocumented JDK internal dependencies, 223
developer-defined annotations, 60
development, 109-118
classpath argument and, 118
command line tools for, 112
Java Development Kit (JDK), 109
Java Runtime Environment (JRE), 109
program structure, 110-112
third-party tools for, 235-237
diamond operator (<>), 164
Diffie-Hellman keys, 104
Digital Signature Algorithm (DSA) generation, 104
direct association of classes, 251
do while loop, 68
Document Object Model (DOM), 105
documentation, Javadoc comments and, 12
domain specific languages (DSLs), 234
Domain Specific Languages (Fowler), 234
double literals, 19
double type, 26
Double wrapper class, 27
DSA generation, 104
durations, 184
Duration.between() method, 184

E

-ea switch, 70
ellipsis (…) in vararg method signatures, 54
empty statements, 64
enableassertions switch, 70
encapsulation, 47
enhanced for loop, 67
entities
floating-point, 26-28
operations involving, 27
enumerations, 59
comparing, 44
declaring in JShell, 204
enum class type, 59
in switch statements, 66
naming conventions for, 4
epoch, 183
equality operators, 41
equals() method (Object), 41
equals() method (String), 42
Era interface, 180
err stream (System), 129
errors, 75
common types of, 77
compile-time errors in JShell snippets, 202
escape sequences, 20
Event Dispatch Thread (EDT), 79
Exception class, 74
exception handling
checked exceptions in JShell, 209
keywords for, 78-82
throw keyword, 78
try-catch statements, 79
try/catch/finally, 78
multi-catch clause, 82
process, 83

programmer-defined exceptions, 83
statements for, 71
try-catch-finally statements, 81
try-finally statements, 80
try-with-resources statements, 82
exceptions, 73-85
checked, 74
common checked/unchecked exceptions and errors, 75-77
errors, 75
hierarchy of, 73
logging, 84
programmer-defined, 83
unchecked, 74
Executor interface, 148
explicit garbage collection, 126
expression statements, 63
extends keyword, 51
extends wildcard, 167

F
fields (see data members)
file I/O, 131-133
reading binary data from files, 132
reading raw character data from files, 131
writing binary data to files, 133
writing character data to files, 132
zipping and unzipping files, 136
FileReader, 131
Files class, 140
Files.newBufferedReader() method, 132
FileVisitor interface, 141
FileWriter, 133
final keyword, 57
final modifier, 89
finalize() method, 126
finally block
in try-catch statements, 79
in try-catch-finally statements, 81
in try-finally statements, 80
float type, 25
Float wrapper class, 27
floating-point entities, 26-28
floating-point literals, 19
fluent APIs, 233-234
fluent interfaces (see fluent APIs)
for loop, 67
enhanced, 67
forward referencing (in JShell), 209
found message (UML), 253
Fowler, Martin, 234, 245
functional interfaces (FIs), 62
annotated, 192
general purpose, 193-195
of Lambda Expressions, 189
@FunctionalInterface annotation, 62, 189

G
G1 collector, 121
garbage collection, 119-121
command-line options for, 122-125
concurrent mark-sweep (CMS), 121
explicit, 126
finalize() method and, 126
Garbage-First (G1) collector, 121
interfacing with, 126
parallel collector, 120
parallel compacting collector, 120
serial collector, 120

Garbage-First (G1) collector, 121
GC (see garbage collection)
generalization of classes, 252
generics, 163-169
  classes and interfaces, 163
  constructors with, 165
  extending, 168
  generic methods in raw types, 169
  get and put principle, 167
  substitution principle, 165
  type parameter names, 5
  type parameters, wildcards, and bounds, 166
get and put principle, 167
getMessage() method (Throwable class), 84
global marking, 121
graphical icon representation, 248
  of classes, 248
  of notes, 248
  of packages, 249
Gregorian calendar, 179
guillemet characters (<< >>), 15, 248
GZIP files, I/O with, 136
GZipInputStream, 137
GZipOutputStream, 137

H

hashCode() method, 42
HashMap() method, 42
HashSet() method, 42
heap, resizing, 125
Heap/CPU Profiling Tool (HPROF), 121
hexadecimal literals, 18
Hijrah calendar system, 180
Horstmann, Cay S., 61
HPROF (Heap/CPU Profiling Tool), 121

I

I/O (input/output), 129-137, 139
  (see also NIO.2)
  class hierarchy for, 130
  with compressed files, 136
  on files, 131-133
  serialization of objects, 135
  sockets, 133-135
  standard streams, in, out, and err, 129
identifiers, 14
  keywords and, 13
IDEs (see integrated development environments)
if else if statement, 65
if else statement, 65
if statement, 64
implements keyword, 58
/imports command (JShell), 200
in stream (System), 129
inconvertible types error, 39
Infinity entity, 26
  operations involving, 27
–Infinity entity, 26
  operations involving, 27
inheritance, 47
  abstract classes and, 55
initializers, static, 57
InputStream, 132
instance variables, 48
  default values for, 36
instances, naming conventions for, 5
Instant.now() method, 183
int type, 25
  in switch statements and, 66
integer literals, 18
Integer wrapper class, 66
integrated development environments, 240
  lambda expressions and, 191
integration libraries (Java), 99
interfaces, 58

declaring in JShell, 204
functional, 62
generic, 163
naming conventions for, 5, 58
intern() method (String), 20
interpreter (Java), 114
InterruptedException, 146
Invocable interface, 172
IOException error, 130
ISO 8601 standard, 179
ISO calendar, 181-186
iteration statements, 66
do while loop, 68
enhanced for loop, 67
for loop, 67
while loop, 68

J

Japanese Imperial calendar system, 180
JAR (see Java Archive utility)
Java
command line tools, 112
compiler, 112-114
interpreter, 114
program structure of, 110-112
Java 8 functional interfaces, 195
Java 8 new features forum at CodeRanch, 195
Java API for XML Web Services (JAX-WS), 106
Java Archive (JAR) utility, 116
executing JAR files, 117
Java Compatibility Kit (JCK), 104
Java Database Connectivity (JDBC), 99, 185
Java Development Kit (JDK), 109
Enhancement Proposals (JEPs), 215
memory management tools, 121
modularization of, 220-222
Java domain specific language (DSL), 234
Java Flight Recorder, 122
Java Generic Security Service (JGSS), 105
Java Generics and Collections (Naftalin, Wadler), 163
Java HotSpot Virtual Machine, 119
Java Mission Control (JMC), 122
Java Naming and Directory Interface (JNDI), 99
Java Object Oriented Querying (jOOQ) API, 233
Java Platform Module System (JPMS), 215
Java Runtime Environment (JRE), 109
Java SE, 95-107
base libraries, 97-99
integration libraries, 99
JavaFX libraries, 100-102
language and utility libraries, 95
Remote Method Invocation (RMI) libraries, 103
security libraries, 104
standard libraries, 95
user interface libraries, 100
XML libraries, 105-107
Java SE 8 for the Really Impatient (Horstmann), 61
Java Shell (see JShell)
The Java Tutorial: Lambda Expressions, 195
Java Virtual Machines (JVMs), 109
garbage collection and, 126
Java Platform Module System (JPMS) implementation, 215
source for, 109
thread priorities and, 145

java.lang package, 59
java.lang.AssertionError, 70
java.lang.NullPointerException, 37
java.lang.Object, 35-46
java.lang.OutOfMemoryError, 125
java.lang.Runnable, 143
java.lang.Thread, 143
java.nio.file.DirectoryStream FI, 142
java.sql package, 185
java.time package, 179
  DateTimeFormatter class, 186
  java.sql and, 185
java.util.concurrent, 148
java.util.function package, 193-195
JavaBeans, 47
Javadoc, comments, 12
JavaFX libraries, 100-102
JAX-WS (see Java API for XML Web Services)
JCK (see Java Compatibility Kit)
JDBC (see Java Database Connectivity)
JDK (see Java Development Kit)
jEdit, 110
JEPs (JDK Enhancement Proposals), 215
JGSS (see Java Generic Security Service)
Jigsaw project, 215
jlink tool, 229
JMC (see Java Mission Control)
jMock API, 233
JNDI (see Java Naming and Directory Interface)
jOOQ API (see Java Object Oriented Querying)
JRE (see Java Runtime Environment)
JRuby, 174
  setting up, 174
JShell (Java Shell), 197-213
  checked exceptions, 209
  dependencies, importation of, 200
  features, 207
  flow control statements and, 199
  forward referencing, 209
  hierarchy and scope, 211
  launching, 197
  method and class declarations, 202
  modifiers, handling, 198
  package declarations and, 199
  primary expressions, 200
  saving, loading, and state, 206
  scratch variables, 207
  snippets, 198
  statements and code blocks, 201
  summary of commands, 212
  tab auto-completion, 208
  viewing, deleting, and modifying snippets, 204
JSR 203 (More New I/O APIs for the Java Platform), 139
JSR 223, 171
  (see also Scripting API)
  scripting languages compatible with, 242
JSR 308 (Type Annotations Specification), 61
JSR 310, 179
  (see also Date and Time API)
JSR 335, 189
JSR 354 (Money and Currency API), 234
JSR 376 (Java Platform Module System), 215
JVisualVM, 125
JVMs (see Java Virtual Machines)

K

keywords, 13
  for exception handling, 78-82

L

lambda expressions, 189-195
  annotated functional interfaces, 192
  community resources for, 195
  example of use, 190
  functional interfaces (FIs), 189
  general-purpose functional interfaces, 193-195
  method and constructor references, 191
  syntax, 190
  tutorials on, 195
language libraries (Java), 95
legacy code, interoperability with Date and Time API, 180
lexical elements, 9-21
  comments, 12
  currency symbols in Unicode, 21
  escape sequences, 20
  identifiers, 14
  keywords, 13
  literals, 17-20
  operators, 15
  separators, 14
  Unicode and ASCII, 9-11
libraries, third-party, 237
lifeline (UML), 254
Lightweight Directory Access Protocol v3 (LDAP), 99
Linux, 109
  POSIX-compliance and, 118
List interface, 153
literals, 17-20
  boolean, 17
  character, 17
  floating-point, 19
  for primitive types, 24
  integer, 18
  null, 20
  string, 19
local variables
  default values for, 36
  final modifier, 88
  naming conventions for, 6
LocalDateTime class, 234
locking threads, 147
locks, applying to blocks and methods, 147
logging exceptions, 84
long integers, 18
long type, 25
looping variables, naming conventions, 7
low-latency collector, 121

M

Mac OS X, 109
  POSIX-compliance and, 118
marker annotation, 60
Maurice Nuftalin's Lambda FAQ, 195
maximum pause time goal, 119
memory management, 119-127
  command-line options for, 122-125
  garbage collection, 119-121
  heap, resizing, 125
  interfacing with the garbage collector, 126-127
  metaspace, 125
  tools for, 121
message to self (UML), 254
metaspace, 125
methods, 48
  abstract, 56, 89
  accessing, 49
  applying locks to, 147
  declaring in JShell, 202
  fluent API prefixes for, 234

generic methods in raw types, 169
interface, 58
in Java 8 and Java 9, 58
invoking, of scripting languages, 172
method references in lambda expressions, 191
naming conventions for, 5
overloading, 49
overriding, 50
passing reference types into, 40
static, 57
vararg, 54-55
Microsoft Windows, 109
file paths, 118
Minquo calendar system, 180
modifiers, 87-90
access, 88
encoding, 90
JShell handling of, 198
nonaccess, 89
module-info.java file, 224
modules, 215-229
compiling, 218
declaring dependencies, 225
defining, 224
defining service providers, 226-229
exporting a package, 224
Java, 216-218
accessibility, 218
automatic modules, 217
JPMS loading behavior, 217
rules for, 216
unnamed, 218
jdeps dependency checker, 222-224
jlink tool, 229
modular JDK, 220-222
naming conventions for, 6
Project Jigsaw, 215
transitive dependencies, 226
Money and Currency API (JSR 354), 234
monitor, 148
multi-catch clause, 82
multidimensional arrays, 38
multiplicity indicators (UML), 249
multivalue annotation, 61
mutator methods, 47
mutex, 148

N

Naftalin, Maurice, 163, 195
naming conventions, 3-7
for acronyms, 3
for annotations, 3
for automatic modules, 217
for classes, 4
for constants, 4
for enumerations, 4
for generic parameter types, 5
for instances, 5
for interfaces, 5
for local variables, 6
for methods, 5
for modules, 6
for packages, 6
for parameters, 6
for static variables, 5
NaN (not a number), 26
operations involving NaN entities, 27
narrowing conversions, 39
Nashorn JavaScript, 173, 177
accessing/controlling Java resources from scripts, 173
native methods, 89
new I/O (NIO) APIs, 129
(see also NIO.2)

newBufferedReader() method (Files), 132
newlines, 20
NIO.2, 139-142
  additional features, 141
  Files class, 140
  Path interface, 139
nonaccess modifiers, 89
not a number (NaN), 26-28
  operations involving NaN entities, 27
Notepad+\+, 110
notes, in UML, 248
notify() method (Object), 146
null literals, 20
Number class, subtypes of, 165
numeric promotion (of primitive types), 28-30
  binary, 29
  special cases for conditional operators, 30
  unary, 29

O

Object class
  equals() method, 41
  methods used for threads, 146
object diagrams (UML), 247
Object Management Group, 245
Object Request Brokers (ORBs), 103
object-oriented programming, 47-62
  annotation types, 59-61
  classes and objects, 47-54
  enumerations, 59
  functional interfaces, 62
  interfaces, 58
ObjectInputStream class, 136
ObjectOutputStream class, 135
objects, 47-54
  accessing methods/data members of, 49
  cloning, 44
  constructors, 51
  copying reference types to, 44
  creating, 48
  deserializing, 136
  methods, 48
  methods overriding, 50
  operators, 15
  overloading methods, 49
  serializing, 135
  this keyword, 53
octal literals, 18
operators, 15
optional software directory, 175
Oracle, 109, 195
Oracle Certified Professional Java SE Programmer Exam, 141
Oracle Java SE Advanced, 122
Oracle Learning Library on YouTube, 195
out stream (System), 129
overloading methods, 49
@Override annotation, 60
overriding methods, 50

P

package-private access modifier, 52, 88
packages
  naming conventions for, 6
  package declarations and JShell, 199
  representing in UML, 249
parallel collectors, 120
parallel compacting collectors, 120
parameters, 163
  (see also type parameters)
  naming conventions for, 6
  naming conventions for generic type, 5
participants (UML), 253
Path interface, 139

PathMatcher interface, 141
Paths class, 139
periods, 184
  Period class, 184
  Period.of() method, 185
Permanent Generation (PermGen) error message, 125
primary expressions, 200
primitive types, 23-33
  autoboxing, 31
  comparison to reference types, 36
  converting between reference types and, 40
  listed, 23
  literals for, 24
  numeric promotion of, 28-30
  passing into methods, 40
  unboxing, 32
  wrapper classes for, 30
printf method as vararg method, 55
println() method, 20
printStackTrace() method (Throwable class), 85
PrintWriter, 132, 134
private access modifier, 88
private data, 47
programmer-defined exceptions, 83
Project Jigsaw, 215
Project Lambda, 189
protected access modifier, 88
protected keyword, 52
public access modifier, 88
publicly available packages, 6
PushbackInputStream class, 132

Q

Queue interface, 153

R

realization models, 252
Red Hat, 109
reference types, 35-46
  comparing, 41-44
    enumerations, 44
    strings, 42
    using equality operators, 41
    using equals() method, 41
  comparing to primitives, 36
  conversion of, 38-40
  converting between primitive types and, 40
  copying, 44-45
    cloning objects, 44
    to an object, 44
  default values of, 36-38
  memory allocation and garbage collection, 46
  passing into methods, 40
regional calendars, 180
Remote Method Invocation (RMI) and CORBA libraries, 103
requires statement, 225
reserved words in statements, 63
resources, accessing/controlling from scripts, 173
Retention meta-annotation, 61
return call (UML), 253
return statement, 70
  JShell and, 199
RMI-IIOP, 103
role names (UML), 250
RSA security interface, 104
run() method (Thread class), 143
Runnable interface, 143
  implementing, 144
Runtime.getRuntime() method, 126
Runtime.getRuntime().gc(), 126
RuntimeException class, 74

S

SASL (see Simple Authentication and Security Layer)
SAX (see Simple API for XML)
Scene Graph API, 101
ScheduledThreadPoolExecutor class, 148
scratch variables, 207
ScriptEngine interface, 171-173
ScriptEngineManager class, 171
Scripting API, 171-177
  script engine implementations, 171-173
    accessing/controlling Java resources from scripts, 173
    embedding scripts in Java, 172
    invoking methods of scripting languages, 172
  scripting languages, 171
  setting up scripting languages and engines, 174-177
scripting engines
  setting up, 174
  validation of, 175
scripting languages, 242
Secured Sockets Layer (SSL), 103
security libraries (Java), 104
self-calls, 254
separators, 14
sequence diagrams (UML), 252
  activation bar, 254
  asynchronous message, 254
  found message, 253
  lifeline, 254
  message to self, 254
  participants, 253
  return call, 253
  synchronous message, 253
serial collectors, 120
Serializable interface, 135
serialization, 45, 135
  deserializing an object, 136
service providers, defining in Java 9, 226
  implementing Service API, 228
  using service providers, 228
Set interface, 153
shallow cloning, 45
short type, 25
  in switch statements and, 66
Short wrapper class, 66
signed types, 24
Simple API for XML (SAX), 106
Simple Authentication and Security Layer (SASL), 105
single abstract method (SAM) interfaces, 62
single value annotation, 60
snippets, 198
  handling thrown exceptions in, 210
  saving, loading, and state, 206
  viewing, deleting, and modifying, 204
socket I/O, 133-135
  reading binary data from sockets, 134
  reading characters from sockets, 133
  writing binary data to sockets, 134
  writing character data to sockets, 134
Solaris, 109
  POSIX-compliance and, 118
SQL (Structured Query Language), 99
  Date Time API and, 185
SSL, 103
stack trace, printing, 85
statements, 63-71
  assert, 70

blocks, 64
conditional, 64-66
empty, 64
exception handling, 71, 78
execution in JShell, 201
expression, 63
iteration, 66-68
synchronized, 70
transfer of control, 68-70
states of threads, 145
static keyword, 57
static modifier, 90
static constants, 57
static data members, 56
static initializers, 57
static methods, 57
static variables, naming conventions for, 5
StAX API (see Streaming API for XML (StAX) API)
Stream API, 141
Streaming API for XML (StAX) API, 106
streams, 129
strictfp, 89
string literals, 19
StringBuffer class, 43
StringBuilder class, 43
strings
compact, 11
comparing, 42
String type in switch statements, 66
Structured Query Language (SQL), 99
Date and Time API and, 185
subclasses, 51-53
substitution principle, 165
super keyword, 52, 167
super wildcard, 167
superclasses, 47, 51-53
Suse, 109
switch statement, 66
synchronized keyword, 147
synchronized methods, 90
synchronized statements, 70
concurrency and, 147
synchronizers, 150
synchronous message (UML), 253
System.err stream, 129
System.gc() method, 126

T

tab auto-completion (JShell), 208
temporary association of classes, 252
temporary variables, naming conventions, 7
testing, third-party tools for, 235-237
TextPad, 110
Thai Buddhist calendar system, 180
third-party tools, 235-244
development, CM, and test tools, 235-237
integrated development environments (IDEs), 240
libraries, 237
scripting languages compatible with JSR 223, 242
web application platforms, 241-242
this keyword, 53
Thread class
extending, 143
methods from, 145
state enumerator, 145
ThreadPoolExecutor class, 148
threads, 143-151
common methods used for
from Object class, 146
from Thread class, 145
Thread class static methods, 147
creating, 143

locking, 147
priorities of, 145
ThreeTen Project, 179
throughput goal, 119
throw keyword, 78
Throwable class, 73
methods providing information on exceptions, 84
throws clause, 74
time, 179
(see also Date and Time API)
Time Zone Database (TZDB), 184
TimeUnit enumeration, 151
timing utility, 151
tools, third-party (see third-party tools)
toString() method (Throwable class), 85
transfer of control statements, 68
break, 68
continue, 69
return, 70
transient data members, 90, 135
transitive dependencies, 226
try-catch statements, 79
try-catch-finally statements, 81
try-finally statements, 80
try-with-resources statements, 82
try/catch/finally blocks, 78
Type Annotations Specification (JSR 308), 61
type parameters (in generics), 163
bounds and wildcards applied to, 166
for generic methods called in non-generic types, 169
naming conventions for, 5
unbounded, 166
types
primitive types and their wrapper classses/reference types, 23-33
reference, 35-46
/types command in JShell, 204

U

Ubuntu, 109
UML Distilled (Fowler), 245
unary numeric promotion, 29
unboxing, 32
Boolean types, 64
unchecked exceptions, 74
common types of, 76
programmer-defined, 84
Unicode, 9-11
compact strings feature, 11
currency symbols in, 21
string literals, 19
Unicode Character Code Chart, 10
Unicode 8.0.0, 9
Unicode Consortium, 9
Unified Modeling Language (UML), 245-254
class relationships, 250-252
classes, diagraming, 245-247
connectors, 249
graphical icon representation, 248
multiplicity indicators, 249
object diagrams, 247
role names, 250
sequence diagrams, 252-254
UNIX Epoch, 183
unnamed modules, 218
unsigned types, 24
user interface libraries
JavaFX, 100-102
miscellaneous, 100
utility libraries, 95

V

varargs, 54-55
Vim, 110
visibility indicators (UML), 247

VisualVM, 126
volatile data members, 90

W

W3C's DOM, 107
Wadler, Philip, 163
wait() method (Object), 146
WatchService interface, 141
web application platforms, 241-242
while loop, 68
whole-heap operations, 121
widening conversions, 39
wildcards in generic type parameters, 166
  extends and super wildcards, 167
wrapper classes for primitive types, 30
  automatic conversion of primitive types to, 32
  automatic conversion to primitive types, 32

X

-X options, 114
X500 Principal Credentials, 105
X500 Private Credentials, 105
XML libraries (Java), 105-107
XML Schema (XSD) types, Date and Time API and, 185
–XX options for garbage collection, 125

Y

YouTube, Oracle Learning Library, 195

Z

ZIP files, I/O with, 136
ZipInputStream, 136
ZipOutputStream, 136

关于作者

Robert James Liguori 是 Oracle 认证的 Java 专家（Oracle Certified Java Professional），作为开发人员参与开发了多个基于 Java 和 Python 的航空航天和自然科学方面的应用程序。

Patricia Liguori 是一位跨学科的信息系统工程师，主要在 The MITRE Corporation（*https://www.mitre.org/*）从事空中交通管制领域的工作。

封面介绍

《Java 口袋指南》封面上的动物是爪哇虎（*Panthera tigris sondaica*）。在近代以来的历史中，这个已灭绝的物种主要生活在印度尼西亚的爪哇岛上，但是化石记录表明，在 12000 年前，它们还生活在婆罗洲岛和巴拉望群岛。

它们的特征是长鼻子、小型骨架以及相对强壮有力的爪子，这些老虎以野猪、鹿和野牛为生。荷兰作家 J.G. ten Bokkel 在 1890 年曾经提到，当提到动物的时候，当地居民会使用一个尊敬的头衔（“Mister Tiger”），因为他们害怕用熟悉的方式来讨论动物会点燃它的怒火。

尽管人们曾经为了保护这些老虎的领地而努力，但是随着爪哇岛人口数量的增长，狩猎、工业发展以及当地内战等因素组合起来，最终导致爪哇虎在 1994 年灭绝了。

O’Reilly 图书封面上的很多动物都濒临灭绝。读者如果想了解能够采取哪些帮助措施，可以访问 *animals.oreilly.com*。

封面图片来源于 Dover Pictoral Archive，封面字体是 URW Typewriter 和 Guardian Sans。文本字体是 Adobe Minion Pro，标题字体是 Adobe Myriad Condensed，代码字体是 Dalton Maag 的 Ubuntu Mono。